Structures de données et algorithmes fondamentaux

Anthony Labarre

Année académique 2019–2020

UNIVERSITÉ PARIS-EST MARNE-LA-VALLÉE

iut Marne-la-Vallée

Avant-propos

Ces notes de cours sont destinées à l'usage des étudiants de première année en DUT Informatique de l'IUT de Champs sur Marne. Elles couvrent la matière vue dans le cours M1103, et probablement un petit peu plus que cela.

Le sujet principal abordé dans ce cours est l'élaboration d'algorithmes efficaces. On y verra donc quelques techniques qui nous permettront d'y arriver, mais les intuitions ne peuvent se développer que par la pratique et les séances d'exercices seront donc essentielles. Pour s'entraîner, il est d'ailleurs conseillé de tenter d'abord de résoudre par soi-même les problèmes énoncés dans ces notes de cours plutôt que de lire directement les solutions et de tenter de les apprendre par cœur sans les comprendre. Même les tentatives infructueuses permettent de développer la réflexion et de pousser la compréhension plus loin.

Quelques exercices supplémentaires sont disséminés dans le texte et vous permettront de vérifier votre compréhension de la matière couverte à mesure de votre progression.

Table des matières

Liste des algorithmes

Chapitre 1

Introduction

Sommaire

1.1 Notions de base

Comme son nom l'indique, l'algorithmique est l'étude des algorithmes.

Définition 1. Un **algorithme** est une méthode de résolution d'un problème en un temps fini, décrite dans un langage suffisamment précis pour qu'il soit traduisible *aisément* et *sans ambiguïté* en une suite d'instructions compréhensibles par une machine.

⚠ Un algorithme n'est pas un programme ! L'algorithme décrit une méthode qui sera ensuite implémentée dans un langage de programmation ; la manière exacte dont cette traduction se fera dépend de plusieurs facteurs, ce qui implique que tous les cas de figures suivants sont possibles :

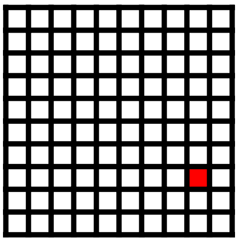
— un algorithme peut être implémenté par une ou plusieurs fonctions ;

— une fonction peut exécuter plusieurs algorithmes ;

— un programme peut contenir plusieurs algorithmes ou fournir un résultat nécessaire à l'exécution d'un autre algorithme ;

— ...

Le travail du concepteur d'algorithmes est donc d'expliquer (à un ordinateur) comment résoudre efficacement un problème. Ceci implique de préciser son mode de raisonnement, car certains problèmes évidents pour un humain le sont moins pour un ordinateur. Par exemple, un humain à qui l'on demande de trouver la case rouge dans la grille ci-dessous y parviendra instantanément :



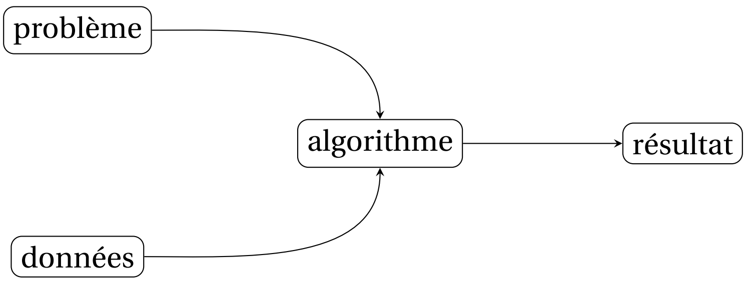
Si l'on pose la même question à un ordinateur, qui ne "voit" pas la solution, il faudra lui expliquer ce qu'on cherche et comment trouver une solution, et donc décrire une méthode de recherche de cette case. Un raisonnement qui se rapprocherait d'une solution satisfaisante serait par exemple :

```
# parcourir les n lignes du tableau
pour i allant de 0 à n-1:
    # parcourir les p colonnes de chaque ligne
    pour j allant de 0 à p-1:
        si tableau(i, j) est rouge:
            renvoyer i et j
```

La solution ci-dessus est formulée en **pseudocode**, un langage fictif qui n'est pas un langage de programmation mais dont la structure en est suffisamment proche pour qu'on puisse facilement traduire ce raisonnement dans le langage de notre choix.

1.1.1 Problèmes et instances

On peut voir un algorithme comme une machine qui va prendre en entrée la description d'un problème et une **instance** de ce problème, c'est-à-dire des données, et qui fournira en sortie une solution à l'instance du problème examiné :



Pour illustrer ces notions de "problème" et d'"instance", prenons l'exemple d'un système de navigation GPS :

— le *problème* typique à résoudre serait, étant données deux villes et un réseau routier, de trouver un chemin le plus court possible entre ces deux villes dans le réseau donné ;

— une *instance* du problème serait un triplet avec des valeurs spécifiques ; par exemple : "départ = Paris, arrivée = Lyon, réseau = routes françaises".

Le format suivant est souvent utilisé pour décrire les problèmes à résoudre :

NOM DU PROBLÈME (exemple : RECHERCHE D'ITINÉRAIRE)	
Données:	les données sur lesquelles on travaille (exemple : une ville de départ, une ville d'arrivée, et un réseau routier).
Question:	le problème à résoudre (exemple : quel est le plus court chemin dans le réseau donné entre les villes données ?)

Au lieu d'une question, on peut également spécifier un résultat (par exemple : la séquence qu'on a reçue dans les données doit devenir triée), ou des objectifs divers. Ce qui importe est que les données et les objectifs soient parfaitement définis.

1.1.2 Correction et efficacité

Deux aspects nous intéressent principalement en algorithmique : on veut que l'algorithme proposé soit :

1. **correct**, c'est-à-dire qu'il donne toujours la bonne réponse *quelles que soient les données d'entrée*; et
2. **efficace**, c'est-à-dire économe en temps de calcul et en ressources.

Les algorithmes construisent des solutions intermédiaires pas à pas, en suivant une succession d'étapes qui finissent par mener à une solution définitive. Dès lors, les preuves de correction se prêtent souvent à une stratégie de démonstration par induction (ou par récurrence), comme on aura l'occasion de le constater plus loin.

La notion d'efficacité est intuitive : on veut atteindre nos objectifs le plus rapidement possible. Cependant, elle requiert une description plus précise et plus formelle; on verra au chapitre 2 comment la caractériser et comment comparer des algorithmes sur base de cette notion.

1.2 Méthodologies

Il n'existe pas d'algorithme pour créer des algorithmes; il est même impossible d'écrire un algorithme permettant de garantir qu'un algorithme qu'on lui fournirait en entrée se termine toujours [3]. Cela dit, il existe quelques principes généraux que l'on peut suivre; les TDs et TPs vous serviront à acquérir ces principes, dont certains sont exposés ci-dessous. On n'en couvrira que deux dans cette section; d'autres stratégies apparaîtront plus loin dans le cours, et seront plus simples à comprendre sur des exemples bien précis.

1.2.1 Formalisation et objectifs

Le développeur est constamment confronté à des problèmes à résoudre pour lesquels il doit trouver des méthodes de résolution *avant* de se lancer dans l'écriture du code. Si l'on ne sait pas quoi écrire, ce n'est pas la peine de lancer son IDE pour commencer à écrire n'importe quoi. Il nous faut donc nous atteler à trouver un raisonnement qui va nous permettre de passer du problème à résoudre à une méthode de résolution et enfin à du code.

Pour se mettre sur la voie, il est bon de formaliser le problème à résoudre et de préciser les objectifs à atteindre. Illustrons ce procédé sur une tâche simple : décider si une liste L est triée (dans l'ordre croissant).

1. **que signifie "être triée"?** Même si la réponse semble évidente, le simple fait de se poser cette question nous met déjà sur la piste d'un algorithme, car elle nous aide à réfléchir à la manière dont on va expliquer à l'ordinateur comment vérifier qu'une liste est triée. Si l'on a du mal à répondre à la question qu'on se pose, on cherche à construire des exemples simples de listes triées et non triées ; il est évident que `[]` et `[1, 2, 3, 4, 5]` sont triées et que `[1, 2, 3, 2, 5]` ne l'est pas. Cela nous conduit à dire qu'**une liste est triée si et seulement si elle est vide ou si chacun de ses éléments (sauf le dernier) est directement suivi d'un élément au moins aussi grand que lui**.
2. **comment automatiser la vérification?** Il nous faut préciser la notion précédente, de manière à lever toute ambiguïté pour pouvoir ensuite passer du raisonnement au code sans le moindre problème. La relecture de la conclusion précédente révèle que l'on aura besoin des *positions* des éléments de notre liste, puisqu'on passe en revue les éléments qui se suivent deux à deux, et qu'il faudra également faire attention aux cas où un élément n'a pas de successeur. Une formulation mathématique nous permet de réécrire la propriété bien plus précisément comme suit :

$$L \text{ triée } \Leftrightarrow \forall\, 0 \leq i \leq |L| - 2 : \quad L[i] \leq L[i+1].$$

3. **comment en déduire le code?** Si les étapes précédentes ont été réalisées avec suffisamment de soin, passer de la formalisation au code devrait être simple. Dans ce cas précis :
   - on passera en revue les positions de la liste à l'aide d'une boucle sur les valeurs $\{0, 1, 2, \ldots, |L| - 2\}$,
   - et on ne renverra vrai qu'après avoir vérifié **toutes** les positions à tester, puisqu'on a un $\forall$ dans la formulation ci-dessus.

   Une première version qu'on pourrait en déduire serait la suivante :

```
def est_triee(L):
    est_croissante = True
    for i in range(len(L) - 1):
        if not (L[i] <= L[i + 1]):
            est_croissante = False
    return est_croissante
```

On a ainsi obtenu une première solution à notre problème de départ. La première approche est rarement la meilleure, mais on obtient au moins un point de départ qu'on peut éventuellement raffiner par la suite. Dans ce cas précis, on peut obtenir une version plus lisible et plus efficace en prenant la négation de la condition trouvée à l'étape 2 : "L est triée $\Leftrightarrow$ la condition est vraie" devient "L *n'est pas* triée $\Leftrightarrow$ la condition est *fausse*". En appliquant les règles de négation des quantificateurs, on trouve :

$$\begin{aligned} L \text{ n'est pas triée} &\Leftrightarrow \neg(\forall\, 0 \leq i \leq |L| - 2 : L[i] \leq L[i+1]) \\ &\Leftrightarrow \exists\, 0 \leq i \leq |L| - 2 : \neg(L[i] \leq L[i+1]) \\ &\Leftrightarrow \exists\, 0 \leq i \leq |L| - 2 : L[i] > L[i+1]. \end{aligned}$$

En supposant la liste triée au départ et en cherchant une contradiction, on obtient donc le code suivant :

```python
def est_triee_2(L):
    for i in range(len(L) - 1):
        if L[i] > L[i + 1]:
            return False
    return True
```

On procède donc à un **parcours** de la liste à la recherche d'une position qui contredit notre hypothèse. Plus généralement, on parlera de "parcours" dans tous les cas où notre algorithme examinera successivement toutes les positions ou tous les éléments d'une structure de données, qu'il s'agisse d'une liste ou d'autre chose.

Les quantificateurs ($\forall$ ou $\exists$) apparaissant dans une définition nous indiquent que l'on aura certainement besoin d'une boucle pour vérifier ce qui nous intéresse : une définition comportant un $\forall$ nous indique qu'il faudra examiner tous les éléments de la structure, tandis qu'une définition comportant un $\exists$ nous indique que l'on pourra s'arrêter dès qu'on aura trouvé ce que l'on cherche.

Exercice 1. Pourquoi la fonction `est_triee_2` est-elle plus efficace que la fonction `est_triee`?

1.2.2 Découpe en sous-problèmes

Dans la vie réelle, on se trouvera souvent amené à résoudre des problèmes beaucoup plus complexes que celui de la section précédente. Il est donc important d'être capable d'identifier des problèmes ou des sous-tâches qui ressemblent à des problèmes qu'on a déjà résolus, voire de réutiliser des algorithmes connus comme des "boîtes noires".

Une méthode simple mais efficace pour résoudre les problèmes auxquels on est confrontés est de les découper en sous-problèmes et de répéter ce procédé jusqu'à ce que les sous-problèmes obtenus soient suffisamment simples à résoudre.

Exemple 1. Mesurons la taille d'un nombre par la longueur de sa factorisation en nombres premiers; par exemple, comme $213 = 3^2 \times 23$, on dit que la taille de 213 est 2 puisque deux nombres premiers apparaissent dans sa factorisation. Supposons qu'on nous donne un ensemble de nombres, et qu'on nous demande de les afficher par taille croissante; pour ce faire, il nous faudra :

1. pouvoir trier des nombres en fonction de leur taille, ce qui implique d'élaborer un algorithme de tri et d'être capable de calculer la taille d'un nombre;
2. pour calculer la taille d'un nombre, il faut être capable de calculer sa décomposition en nombres premiers;
3. pour calculer cette décomposition, il nous faut soit connaître les nombres premiers dont on aura besoin, soit être capable de les générer;
4. pour générer les nombres premiers, on devra être capable de vérifier si un nombre donné est premier;

⋮

- *(fin exemple 1)* - -

1.2.3 Correction

Un algorithme doit résoudre le problème donné, **quelles que soient les données à traiter**. Il faut donc s'assurer que l'algorithme proposé donnera *toujours* un résultat correct — et bien entendu, il faut commencer par vérifier que l'algorithme se termine ! Comme en pratique, on ne peut pas le tester sur toutes les entrées possibles puisqu'il y en aura la plupart du temps une infinité, il sera important de pouvoir **démontrer** qu'un algorithme est correct.

Parfois, la correction d'un algorithme est relativement évidente. Quand ce n'est pas le cas, il nous faut trouver le raisonnement qui permettra de le démontrer ; les démonstrations par induction conviendront généralement bien à ce genre de tâche, et consisteront dans les grandes lignes à montrer que les modifications apportées à une solution partiellement correcte nous donnent une nouvelle solution toujours partiellement correcte mais légèrement plus proche du résultat final. Lorsque l'algorithme se terminera, la solution partielle sera devenue la solution complète et sa correction découlera de la preuve par induction.

1.3 Remarques

Ce cours se focalise sur l'algorithmique et aurait pu (ou dû ?) décrire les algorithmes en pseudocode. L'utilisation de Python permettra cependant d'écrire des algorithmes prêts à l'emploi et plus faciles à évaluer en pratique, ce qui nous mènera en contrepartie à devoir de temps en temps expliquer des subtilités qui sont propres à ce langage et qui n'auraient donc normalement pas leur place dans un cours d'algorithmique.

Si un compromis est inévitable, l'accent sera mis dans le code présenté sur la lisibilité plutôt que sur l'efficacité, afin que les algorithmes présentés soient facilement adaptables dans d'autres langages. On réimplémentera également certains algorithmes existant déjà en Python, afin de comprendre leur fonctionnement et de pouvoir les adapter à d'autres problèmes.

Les livres de Skiena [2] et Cormen et al. [1] constituent deux références générales utiles.

Bibliographie

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, et C. Stein, *Introduction to Algorithms*, MIT Press, 3ème édition, 2009.

[2] S. Skiena, *The Algorithm Design Manual*, Springer, 2ème édition, 2008.

[3] A. M. Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proceedings of the London Mathematical Society, s2-42 (1937), pages 230–265.

Chapitre 2

Complexité algorithmique

Sommaire

Outre le fait qu'un algorithme doit être correct, et donc se terminer, on souhaite également que cet algorithme soit le plus efficace possible, c'est-à-dire qu'il doit être rapide (en termes de temps d'exécution) et économe en ressources (espace de stockage, mémoire utilisée). Ce chapitre vise à formaliser ces notions, et à présenter les outils qui nous permettront d'évaluer les performances **théoriques** de nos algorithmes. La sous-section 2.1.2 expliquera pourquoi l'approche théorique est préférable à l'approche pratique ; on verra également que les prédictions réalisées en théorie sont — heureusement ! — fiables en pratique dans la grande majorité des cas.

2.1 Temps d'exécution

On veut qu'un algorithme soit correct, mais aussi **efficace**, c'est-à-dire : rapide (en termes de temps d'exécution) et économe en ressources (espace de stockage, mémoire utilisée). On a donc besoin d'outils qui nous permettront d'évaluer la qualité des algorithmes proposés. Comment faire ?

La première idée qui vient à l'esprit est sans doute de mesurer le temps qu'un algorithme met à s'exécuter. Si on l'a implémenté, on peut utiliser les outils disponibles dans le langage choisi pour chronométrer le morceau de code qui nous intéresse. Il faut cependant être plus précis :

— intuitivement, presque tout algorithme s'exécutant sur des données “petites” sera rapide et sera lent sur des données “grandes”. Comme on ne sait pas *a priori* à partir de quelle taille de données le temps de calcul de l'algorithme devient prohibitif, on préfère l'exécuter plusieurs fois sur des données dont la taille croît progressivement, jusqu'à ce qu'on atteigne ce seuil ou que l'on ait une idée de la manière dont ce temps d'exécution augmente.

— de plus, il ne suffit pas de procéder à une seule exécution de cet algorithme par taille de données, car on pourrait de temps en temps tomber sur des cas où l'algorithme est rapide et d'autres où l'algorithme est plus lent. Une façon d'obtenir des mesures plus réalistes est de l'exécuter plusieurs fois sur des données de même taille, et de conserver le temps moyen d'exécution pour chaque valeur de taille qui nous intéresse.

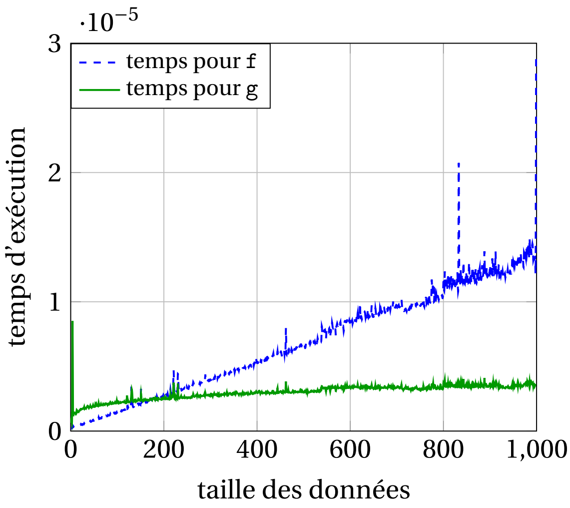
— enfin, il ne suffit pas de faire des moyennes sur n'importe quelles instances : il faudrait également les choisir aléatoirement, encore une fois par souci d'objectivité. Si un algorithme s'exécute rapidement sur un cas particulier qui nous arrange, cela ne nous garantit absolument pas qu'il sera efficace en général.

Exemple 2. Voici deux fonctions vérifiant si un élément appartient à une liste :

```
def f(T, x):
    if x in T:
        return True
    return False
```

```
def g(T, x, a=0, b=0):
    if a > b:
        return False
    p = (a + b) // 2
    if T[p] == x:
        return True
    if T[p] > x:
        return g(T, x, a, p-1)
    return g(T, x, p+1, b)
```

Laquelle choisir ? La réponse n'est pas évidente à première vue. Pour s'aider, on peut comparer le temps d'exécution moyen des deux fonctions sur des listes de plus en plus grandes, en prenant en compte les remarques ci-dessus :



On constate maintenant que la fonction g est en général beaucoup plus rapide que la fonction `f` ; mais cette comparaison ne nous dit pas **pourquoi** cela est vrai.

- *(fin exemple 2)* - -

2.1.1 Mesure du temps d'exécution en Python

Le graphe réalisé dans l'exemple 2 nécessite de savoir comment mesurer le temps d'exécution d'un bout de code. En Python, on peut simuler un chronomètre :

— on le déclenche juste avant le début du bout de code ;

— on l'arrête juste après la fin du bout de code ;

— le temps écoulé entre les deux pressions est la durée qui nous intéresse.

On peut y parvenir soit avec la fonction `time()`, soit avec la fonction `clock()`, toutes deux disponibles dans le module `time`.

Exemple 3. Utilisons la fonction `time()` pour chronométrer un morceau de code dans l'interpréteur Python :

```
>>> from time import time
>>> debut = time()      # enclencher le chronomètre
>>> # morceau de code
>>> fin = time()           # arrêter le chronomètre
>>> print(fin - debut, 'secondes')
2.4835598468780518 secondes
```

Il est rarement utile de connaître toutes les décimales ; si on veut n'en garder que deux, par exemple, on peut :

1. multiplier la mesure par 100 ;
2. tronquer la partie décimale ;
3. diviser le résultat par 100.

En Python, cela donnerait : `int(duree * 100) / 100`. De manière générale, si on veut garder k décimales, on convertit en entier la durée multipliée par 10^k, et on la divise ensuite par 10^k.

Attention : tout comme les tests, les mesures de temps d'exécution ne font jamais partie du "vrai" code, à savoir celui que l'on veut réellement écrire et incorporer dans la version finale de notre programme. Dès lors, on s'arrangera toujours pour les placer à des endroits où l'on peut facilement les retirer.

Exemple 4. Voici deux morceaux de code mesurant le temps d'exécution d'une fonction :

Mauvaise mesure du temps de calcul

```
def ma_fonction(mes_parametres):
    debut = time()
    # ...
    fin = time()
    return fin - debut

ma_fonction(parametres)
```

Bonne mesure du temps de calcul

```
def ma_fonction(mes_parametres):
    # ...


debut = time()
ma_fonction(parametres)
fin = time()
```

2.1.2 Calcul théorique du temps d'exécution

La mesure en pratique du temps d'exécution de morceaux de code que l'on a déjà programmés est utile, mais pas entièrement satisfaisante pour plusieurs raisons. Parmi elles :

1. les mesures obtenues ne sont valides que pour une certaine machine à un moment donné dans un état bien précis [1]. Dès lors, les résultats ne sont pas nécessairement transposables à une autre machine basée sur un matériel différent.
2. cette technique est utile lorsqu'on a déjà écrit le code correspondant à un algorithme ; mais si l'algorithme est un tant soit peu complexe, rédiger — et débugger — le code correspondant et le tester de manière pertinente peut prendre beaucoup de temps.

C'est pourquoi on désirerait disposer d'un procédé qui nous permettrait d'évaluer l'efficacité d'un algorithme *sans l'implémenter* et *indépendamment de l'architecture de la machine*.

Dès lors, on ne mesurera pas la durée en heures, minutes, secondes, ... Au lieu de cela, on utilisera des **unités de temps abstraites** proportionnelles au nombre d'opérations effectuées. Au besoin, on pourra ensuite adapter ces quantités en fonction de la machine sur laquelle l'algorithme s'exécute.

On applique les règles ci-dessous pour calculer le temps d'exécution d'un algorithme :

— chaque **instruction basique** (affectation d'une variable, comparaison, $+$, $-$, $*$, $/$, ...) consomme une unité de temps ;

— chaque **itération** d'une boucle rajoute le nombre d'unités de temps consommées dans le corps de cette boucle ;

— chaque **appel de fonction** rajoute le nombre d'unités de temps consommées dans cette fonction ;

— pour avoir le nombre d'opérations effectuées par l'algorithme, on additionne le tout.

1. Pour vous en convaincre, effectuez plusieurs fois exactement la même mesure sur votre propre machine ; rares seront les cas où vous obtiendrez deux fois le même résultat.

Exemple 5. Calculons le temps d'exécution de la fonction suivante, qui renvoie la factorielle de n, c'est-à-dire $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$ (avec $0! = 1$) :

```
def factorielle(n):
    fact = 1                    # initialisation: 1
    i = 2                       # initialisation: 1
    while i <= n:               # itérations: au plus n-1
        fact = fact * i         # multiplication + affectation: 2
        i = i + 1               # addition + affectation: 2
    return fact                 # renvoi d'une valeur: 1
```

On a aussi un test à chaque itération à prendre en compte. Le nombre total d'opérations est donc :

$$1 + 1 + (n-1) * 5 + 1 + 1 = 5n - 1$$

Remarquons que ce genre de calcul n'est pas tout à fait exact :

— on ne sait pas toujours combien de fois exactement on va effectuer une boucle ;

— de même, lors d'un branchement conditionnel, le nombre de comparaisons effectuées n'est pas toujours le même ; par exemple, si `a` est faux dans le test `a and b`, alors évaluer `b` est inutile puisque `a and b` sera toujours faux, et Python ne le fera donc pas.

Heureusement, on dispose d'outils plus simples à manipuler et plus adaptés.

2.2 La notation $O(\cdot)$

La **théorie de la complexité** (algorithmique) vise à répondre à ces besoins. Elle permet :

1. de classer les problèmes selon leur difficulté ;
2. de classer les algorithmes selon leur efficacité ;
3. de comparer les algorithmes <u>sans</u> devoir les implémenter.

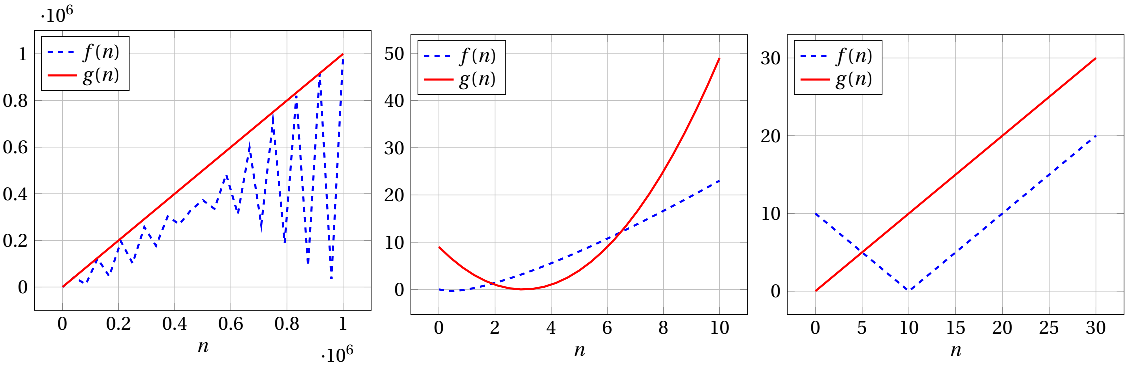
Comme on l'a vu dans la section précédente, les calculs à effectuer pour évaluer le temps d'exécution d'un algorithme peuvent parfois être longs et pénibles. De plus, le degré de précision qu'ils requièrent est souvent inutile. On aura donc recours à une *approximation* de ce temps de calcul, représentée par la notation $O(\cdot)$.

Définition 2. Une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

$$\exists\, n_0 \in \mathbb{N}, \exists\, c \in \mathbb{R}, \forall\, n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

Autrement dit : $f(n)$ est en $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f(\cdot)$ est toujours dominée par la fonction $g(\cdot)$, à une constante multiplicative fixée près. Les valeurs absolues importent dans la définition, mais on verra qu'en pratique, on ne devra jamais s'en soucier ; les fonctions qui nous intéresseront mesureront toujours le temps d'exécution d'un algorithme, qui sera forcément positif.

Exemple 6. Voici quelques cas où $f(n) = O(g(n))$:



Pour prouver qu'une fonction f est en grand O d'une fonction g, on peut s'appuyer sur la définition comme le montre l'exemple suivant. En pratique, on aura plutôt recours aux règles de calcul et aux simplifications exposées dans la section 2.3.

Exemple 7. Utilisons la définition 2 pour prouver que la fonction $f_1(n) = 5n + 37$ est en $O(n)$:

1. notre but est de trouver une constante $c \in \mathbb{R}$ et un seuil $n_0 \in \mathbb{N}$ à partir duquel $|f_1(n)| \leq c|n|$.
2. on remarque que $|5n + 37| \leq |6n|$ si $n \geq 37$:
   $|5 * 37 + 37| \leq 6 * |37|$;
   $|5 * 38 + 37| \leq 6 * |38|$;
   $\vdots$
3. on en déduit donc que $c = 6$ fonctionne à partir du seuil $n_0 = 37$, et on a fini.

Remarque : on ne demande pas d'optimisation (le plus petit c ou n_0 qui fonctionne), juste de donner des valeurs qui fonctionnent ; $c = 10$ et $n_0 = 8$ sont donc aussi acceptables.

2.3 Calcul de la complexité d'un algorithme

La notation $O(\cdot)$ va nous permettre de quantifier l'efficacité d'un algorithme sous certaines hypothèses :

Définition 3. La **complexité** d'un algorithme est la mesure **asymptotique** de son temps d'exécution **dans le pire cas**. Elle s'exprime à l'aide de la notation $O(\cdot)$ en fonction de la taille des données reçues en entrée.

Les deux précisions sur le caractère de cette mesure importent :

1. **asymptotique** signifie que l'on s'intéresse à des données très grandes ; en effet, les petites valeurs ne sont pas assez informatives ;
2. **"dans le pire cas"** signifie que l'on s'intéresse à la performance de l'algorithme dans

les situations où le problème prend le plus de temps à résoudre ; on veut être sûr que l'algorithme ne prendra jamais plus de temps que ce qu'on a estimé.

Pour voir ce qui justifie l'étude du comportement asymptotique des fonctions, regardons les deux graphiques de la Figure 2.1. Ils comparent les deux mêmes fonctions mesurant le temps d'exécution de deux algorithmes en fonction de la croissance de la taille des données qu'ils manipulent ; la partie (a) nous mène à croire que la courbe verte semble correspondre à un algorithme beaucoup plus efficace, mais la partie (b) nous montre toutes les informations nécessaires pour faire le bon choix.

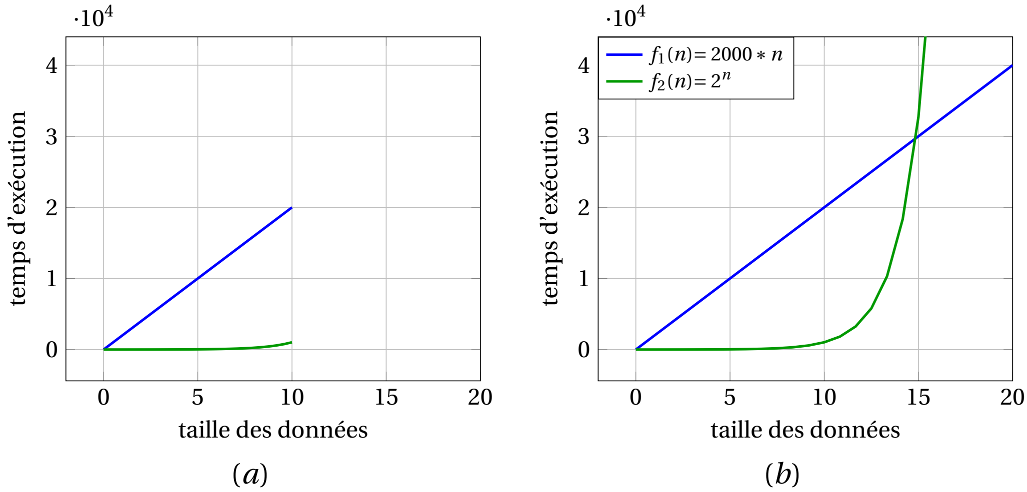


FIGURE 2.1 – Comparaison des temps d'exécution de deux algorithmes avec (a) peu d'informations et (b) suffisamment d'informations.

2.3.1 Les bases

Le calcul de la complexité d'un algorithme s'effectue de manière similaire à celui du temps d'exécution vu précédemment, si ce n'est qu'on remplace maintenant les unités de temps par les approximations fournies par la notation $O(\cdot)$. Plus précisément :

— chaque instruction basique (affectation d'une variable, comparaison, $+$, $-$, $*$, $/$, ...) consomme un **temps constant**, représenté par la notation $O(1)$;

— chaque itération d'une boucle rajoute la complexité de ce qui est effectué dans le corps de cette boucle ;

— chaque appel de fonction rajoute la complexité de cette fonction ;

— pour obtenir la complexité de l'algorithme, on additionne le tout.

On aura aussi recours aux simplifications suivantes :

1. on oublie les constantes multiplicatives (elles valent 1) ;
2. on annule les constantes additives ;
3. on ne retient que les termes dominants.

Exemple 8 (simplifications)**.** Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations;

1. on remplace les constantes multiplicatives par 1 : $\qquad 1n^3 - 1n^2 + 1n + 3$
2. on annule les constantes additives : $\qquad n^3 - n^2 + n + 0$
3. on garde le terme de plus haut degré : $\qquad n^3 + 0$

et on a donc $g(n) = O(n^3)$.

2.3.2 Combinaison des complexités

On additionne les complexités d'opérations en séquence; donc $O(f_1(\cdot)) + O(f_2(\cdot)) = O(f_1(\cdot) + f_2(\cdot))$. On applique le même traitement aux branchements conditionnels :

```
if <condition>:
    # instructions (1)
else:
    # instructions (2)
```

$$\left.\begin{array}{l} O(g(\cdot)) \\ O(f_1(\cdot)) \\ \\ O(f_2(\cdot)) \end{array}\right\} = O(g(\cdot) + f_1(\cdot) + f_2(\cdot))$$

En particulier, une somme d'opérations constantes sera constante : $O(1) + O(1) = O(1)$.

Dans les boucles, on multiplie la complexité du corps de la boucle par le nombre d'itérations. La complexité d'une boucle `while`[2] se calcule comme suit :

```
# en supposant qu'on a m itérations
while <condition>:
    # instructions
```

$$\left.\begin{array}{l} O(g(\cdot)) \\ O(f(\cdot)) \end{array}\right\} = O(m * (g(\cdot) + f(\cdot)))$$

Si deux boucles se suivent, il s'agit de blocs d'opérations en séquence : on additionnera donc leurs complexités. En revanche, si la boucle 1 contient la boucle 2, alors chaque itération de la boucle 1 nous fera exécuter *toutes* les itérations de la boucle 2, et il faudra donc *multiplier* leurs complexités.

2.3.3 Application des règles

Pour calculer la complexité d'un algorithme :

1. on calcule la complexité de chaque "partie" de l'algorithme;
2. on combine ces complexités conformément aux règles qu'on vient de voir;
3. on simplifie le résultat grâce aux règles de simplifications qu'on a vues :
   — élimination des constantes, et
   — conservation du (des) terme(s) dominant(s).

Exemple 9 (factorielle de n)**.** Reprenons le calcul de la factorielle, qui nécessitait $5n - 1$ opérations :

```python
def factorielle(n):
```

2. Un raisonnement analogue fonctionne pour les boucles `for`.

```
    fact = 1              # initialisation: O(1)
    i = 2                 # initialisation: O(1)

    while i <= n:         # n-1 itérations: O(n)
        fact = fact * i   # multiplication: O(1)
        i = i + 1         # incrémentation: O(1)

    return fact           #         renvoi: O(1)
```

Conformément aux règles vues plus haut, la complexité de la fonction est $O(1)+O(n)*O(1)+O(1)=O(n)$.

--- *(fin exemple 9)* --

2.3.4 Justification des simplifications

Les simplifications vues dans la sous-section 2.3.1 nous arrangent, mais sont-elles justifiées ? Commençons par l'attribution d'un même coût constant aux opérations basiques : même si en général, une multiplication coûtera plus cher qu'une addition, on ne sait pas a priori sur quelle machine l'algorithme s'exécutera, et l'on ne veut pas non plus faire d'hypothèse à ce sujet. Il est donc raisonnable de les classer dans la même catégorie des opérations en temps constant ; *ce qui ne signifie pas nécessairement que cette constante est la même pour toutes ces opérations.*

En ce qui concerne l'oubli des constantes multiplicatives, l'annulation des constantes additives, et l'élimination des termes non dominants, ces trois règles de simplifications viennent du fait qu'on s'intéresse à la croissance du temps d'exécution en fonction de la taille des données. Même si une constante est relativement grande, sa taille ne changera par définition pas et finira par devenir négligeable par rapport à la taille des données au fur et à mesure que celle-ci croît. Ceci nous permet de nous concentrer sur les caractéristiques de l'algorithme qui seront les plus coûteuses en termes de temps d'exécution, et de pouvoir facilement décider si un algorithme fonctionnera de manière acceptable sur des données de très grande taille.

C'est pourquoi on décide d'éliminer ces termes moins importants, car même s'ils auront un impact en pratique, ce n'est en général pas eux qui contribueront le plus à la croissance du temps d'exécution de l'algorithme. On préfère donc avoir une **idée** de cette croissance plutôt qu'une expression plus précise mais inutilement compliquée. Il faut garder à l'esprit que la complexité est un outil parmi d'autres qui nous permettra de faire un choix entre nos algorithmes ; mais dans certaines situations, il nous faudra pousser l'analyse plus loin pour départager des algorithmes semblant équivalents.

2.3.5 Calcul de la complexité grâce aux limites

Lorsqu'on doit classer deux fonctions $f(n)$ et $g(n)$ en termes de notation $O(\cdot)$, la première étape consiste à appliquer les règles de simplifications et à voir si les fonctions simplifiées sont plus faciles à comparer. Il existe cependant des cas où l'on ne voit pas directement, même sur les fonctions simplifiées, laquelle domine l'autre en termes de croissance asymptotique. Dans ce cas, il est utile de procéder à un calcul de limite : intuitivement, si la croissance de la fonction $g(n)$ domine celle de $f(n)$ pour des grandes valeurs de n, alors la valeur de $\lim_{n\to+\infty} f(n)/g(n)$ doit être une constante (éventuellement nulle).

Exemple 10. Quelle fonction croît le plus rapidement : $\ln n$ ou $\sqrt{n}$? Pour le déterminer, calculons $\lim_{n\to+\infty} \ln n/\sqrt{n}$; en appliquant la règle de l'Hospital, on obtient

$$\lim_{n\to+\infty} \frac{\ln n}{\sqrt{n}} \overset{RH}{=} \lim_{n\to+\infty} \frac{1/n}{1/2\sqrt{n}} = \lim_{n\to+\infty} \frac{2\sqrt{n}}{n}.$$

Cette limite tend vers 0 (ce que l'on peut prouver par le même procédé), et l'on en conclut donc que $O(\ln n)$ est inférieure à $O(\sqrt{n})$.

2.4 Classifications

2.4.1 Classification d'algorithmes

La relation “être en grand O de ...” est réflexive, mais pas symétrique. En effet :

— $f(n) = O(g(n)) \not\Rightarrow g(n) = O(f(n))$: par exemple : $5n + 43 = O(n^2)$, mais $n^2 \neq O(n)$;

— $f(n) \neq O(g(n)) \not\Rightarrow g(n) \neq O(f(n))$: par exemple : $18n^3 - 35n \neq O(n)$, mais $n = O(n^3)$.

Lorsque la relation est symétrique, on considère les deux fonctions équivalentes.

Définition 4. Deux fonctions $f(n)$ et $g(n)$ sont **équivalentes** du point de vue de la notation $O(\cdot)$ si $f(n) = O(g(n))$ et $g(n) = O(f(n))$.

On peut ainsi “ranger” les fonctions équivalentes dans une même **classe**. La Figure 2.2 montre quelques classes de complexité fréquentes (par ordre croissant en termes de $O(\cdot)$).

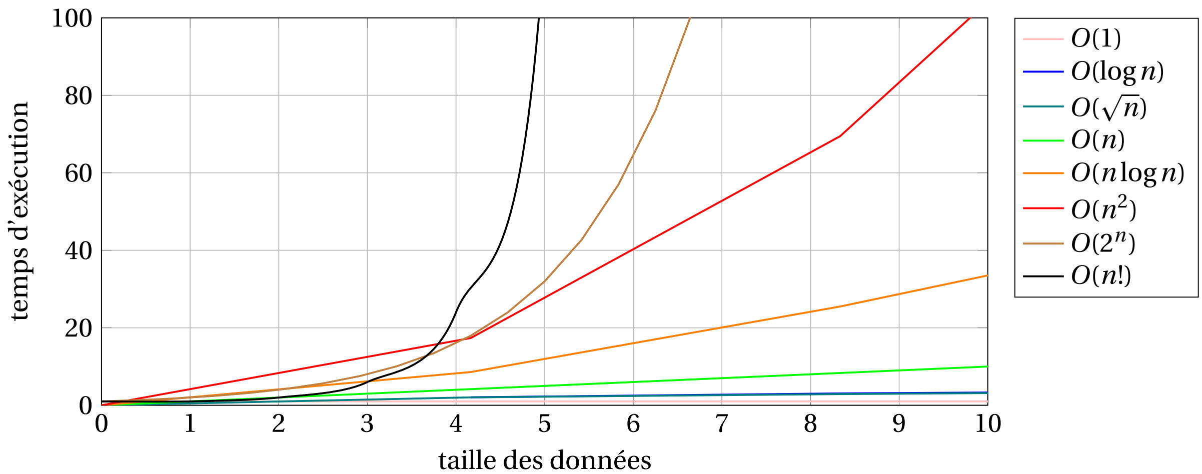


FIGURE 2.2 – Quelques classes de complexité fréquentes.

D'un point de vue algorithmique, trouver un nouvel algorithme de même complexité pour un problème donné présente en général un intérêt limité. Bien entendu, il arrive que l'omission des termes cachés par la notation $O(\cdot)$ nous réserve des surprises en pratique, et on verra plus loin des exemples d'algorithmes de même complexité mais dont les performances en pratique seront très différentes. Mais le critère fourni par la notation $O(\cdot)$ est suffisamment fiable pour nous permettre de réaliser de bonnes prédictions, et nous donner un critère simple de choix en pratique :

On choisit l'algorithme dont la complexité est la plus faible.

Le Tableau 2.1 montre la croissance du temps d'exécution en fonction de la taille des données sur base d'unités concrètes de temps et permet de mieux se rendre compte des impacts pratiques.

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

TABLE 2.1 – Comparaison en pratique du temps de calcul qu'impliquent les différentes complexités [2], en supposant qu'une opération s'exécute en une nanoseconde.

Pour faire un choix éclairé entre plusieurs algorithmes, il faut être capable de situer leur complexité. On fait une première distinction entre les deux classes suivantes :

1. les algorithmes dits **polynomiaux**, dont la complexité est en $O(n^k)$ pour une certaine **constante** k ; si k fait partie des données que l'algorithme reçoit, il s'agit par définition d'une *variable* et l'on ne pourra donc pas parler de complexité polynomiale ;
2. les algorithmes dits **exponentiels**, dont la complexité ne peut pas être majorée par une fonction polynomiale. Par exemple, une fonction en $O(n!)$ sera considérée comme exponentielle, même si cela ne correspond pas à la définition vue dans les cours de mathématiques.

2.4.2 Classification de problèmes

On peut calculer la complexité d'un algorithme, mais également celle d'un problème. La complexité d'un problème est celle du meilleur algorithme connu pour le résoudre, et ce critère peut également être utilisé pour classer les problèmes plutôt que les algorithmes :

1. un problème de complexité polynomiale est considéré "facile" ;
2. sinon (complexité non-polynomiale ou inconnue (!)), il est considéré "difficile".

La classification de problèmes présente plusieurs intérêts :

1. **"recyclage" de résultats :** on peut parfois montrer que deux problèmes donnés sont équivalents, ou que l'on peut résoudre l'un à l'aide de l'autre. Par conséquent :

— si l'on peut résoudre le problème P_1 en résolvant le problème P_2, alors on a directement un algorithme pour P_1 grâce à P_2 ;

— on a donc complexité(P_1) $\leq$ complexité(P_2), et on ne peut donc pas résoudre P_2 plus vite que P_1.

Ce second point nous permet parfois de prouver qu'un algorithme est *optimal* : par exemple, si l'on en déduit que notre problème ne peut pas se résoudre en temps inférieur à $O(n^2)$ et que l'on a trouvé un algorithme en $O(n^2)$, alors on sait que ce n'est pas la peine de chercher un algorithme de complexité inférieure puisqu'il ne peut pas en exister, et notre algorithme est donc optimal [3].

2. **déterminer la stratégie à utiliser :** certains problèmes sont particulièrement difficiles, et on ne peut parfois pas espérer les résoudre en temps polynomial (cf. plus loin). Si l'on peut prouver qu'un problème appartient à cette catégorie, on a une meilleure idée des techniques qui ont une chance de marcher pour le résoudre en pratique ainsi que du réalisme des exigences que l'on peut avoir vis-à-vis des solutions recherchées. On se rendra compte que certains problèmes nous obligeront à être patients, ou à accepter des solutions non optimales.

3. **applications en cryptographie :** les problèmes difficiles à résoudre ont leur utilité en cryptographie, où ils permettent de garantir la confidentialité des données en s'appuyant justement sur le fait que pour déchiffrer ces messages, il faudrait résoudre un problème difficile. Citons par exemple le **cryptosystème de Rabin**, inventé en 1979 par Michael O. Rabin : il s'appuie sur le problème de factorisation, qu'on suppose difficile. Il s'agit d'un cryptosystème **asymétrique** : on chiffre les données avec une **clé publique**, à laquelle tout le monde a donc accès, et on déchiffre les données à l'aide d'une **clé privée** que seules les personnes de confiance possèdent.

   (a) pour générer les clés, on choisit deux grands nombres premiers p et q, qui forment la clé privée ; et la clé publique sera le nombre $n = p \times q$;

   (b) le chiffrement des données s'effectue en prenant le carré des entrées modulo n (les caractères sont donc interprétés comme des chiffres) ;

   (c) et le déchiffrement des données s'effectue en calculant les racines carrées modulo p et q.

   Ainsi, si l'on connaît la clé privée, le déchiffrement est facile ; sinon, il faut factoriser n ... mais on ne connaît pas d'algorithme de complexité polynomiale en b (le nombre de bits de n), et rien ne garantit qu'un tel algorithme existe [4].

2.5 Digression : P, NP, $

Il existe en informatique théorique une question fondamentale liée à la complexité des problèmes. Comme on l'a vu dans la section précédente, on peut classer les problèmes en fonc-

3. Optimal *du point de vue de la complexité* ; il peut bien sûr exister des optimisations à apporter en pratique, mais ceci relèvera plutôt du langage de programmation choisi.

4. L'algorithme passant en revue tous les diviseurs jusqu'à n en examine $O(n)$, mais comme on a besoin de $\log_2 n$ bits pour encoder n, la complexité exprimée en termes de b est $O(2^b)$ puisque $n = 2^{\log_2 n}$ et la complexité s'exprime en fonction de la *taille* des données.

tion de leur difficulté ... mais ceci requiert évidemment de connaître des algorithmes permettant de les résoudre ou de pouvoir garantir qu'on ne peut pas faire mieux qu'une certaine complexité.

Il n'est pas toujours possible de garantir qu'un problème n'est pas soluble en temps polynomial, et une certaine catégorie de problèmes, qu'on qualifie de NP-complets, sont particulièrement difficiles. Les détails et la définition formelle de ces classes de problèmes sortent largement du cadre de ce cours, mais on peut se faire une idée informelle d'une des questions les plus importantes de l'informatique théorique en décrivant les deux classes suivantes de problèmes :

— **la classe** P est l'ensemble des problèmes qu'on peut résoudre avec un algorithme de complexité polynomiale ;

— **la classe** NP est l'ensemble des problèmes dont on peut **vérifier** une solution avec un algorithme de complexité polynomiale.

La question non-résolue (depuis maintenant plusieurs décennies) que se posent les théoriciens est :

Est-ce que P=NP **?**

Dans l'affirmative, cela signifierait que tout problème dont on peut vérifier une solution en temps polynomial peut également être résolu en temps polynomial ; ou dit autrement : s'il est "facile" de vérifier une solution, il est également "facile" d'en trouver une. Cette question est tellement importante que le prestigieux *Clay Mathematics Institute* offre un million de dollars pour sa résolution [5].

Il est naturel de se demander ce qui fait la difficulté d'un problème, et contrairement à ce que l'intuition d'un néophyte suggère, la difficulté d'un problème est rarement corrélée avec celle de sa description. Quelques exemples de problèmes faciles qu'on examinera sous diverses formes dans ce cours sont :

— la recherche d'un / le plus petit / le plus grand élément dans une base de données ;

— le tri d'une base de données selon divers critères ;

— rechercher une / plusieurs / toutes les occurrences d'un motif dans un texte ;

— calculer le plus court chemin entre deux villes dans un réseau routier ;

— ...

Le problème suivant ne semble par contre ni difficile à comprendre, ni difficile à résoudre ; pourtant, il appartient à la classe des problèmes dits "NP-complets", perçus comme difficiles, et qui jouent un rôle central dans la question "P ?=NP" :

SUBSET SUM	
Données:	un ensemble S de n entiers ;
Question:	existe-t-il un sous-ensemble $S' \subseteq S$ dont la somme des éléments est nulle ?

Une instance de SUBSET SUM pourrait être $S = \{-7, -3, -2, 5, 8\}$; dans ce cas-ci, le sous-ensemble $S' = \{-3, -2, 5\}$ est une solution. On voit bien qu'il est facile de vérifier en temps polynomial que S' est une solution, mais pas comment la trouver.

5. Voir http://www.claymath.org/millennium/P_vs_NP/.

Voici un autre exemple de problème NP-complet d'apparence inoffensive :

PARTITION	
Données:	un ensemble S de nombres (répétitions autorisées) ;
Question:	peut-on séparer S en deux sous-ensembles A et B tels que $\sum_{a\in A} a = \sum_{b\in B} b$?

Une instance de PARTITION pourrait être $S = \{1,3,1,2,2,1\}$, et une solution pour cette instance serait $A = \{1,3,1\}$ et $B = \{1,2,2\}$. Là encore, trouver un algorithme vérifiant notre solution en temps polynomial est trivial, mais trouver une solution s'avère bien plus complexe.

Enfin, mentionnons qu'il existe des problèmes dont la complexité est inconnue : on ne sait pas s'ils sont dans P, s'ils sont NP-complets ... ou autre chose. Le problème suivant appartient à cette catégorie : il traite de structures qu'on appelle des **graphes** (des points appelés "sommets" reliés par des liens appelés "arêtes") et demande simplement si deux graphes donnés sont les mêmes :

ISOMORPHISME DE GRAPHES	
Données:	deux graphes G_1 et G_2 ;
Question:	G_1 et G_2 sont-ils "les mêmes" ? (peut-on numéroter leurs sommets de manière à obtenir deux ensembles d'arêtes identiques ?)

La Figure 2.3 illustre trois instances de ce problème. Les réponses sont évidentes dans les deux premiers cas ... moins dans le troisième[6].

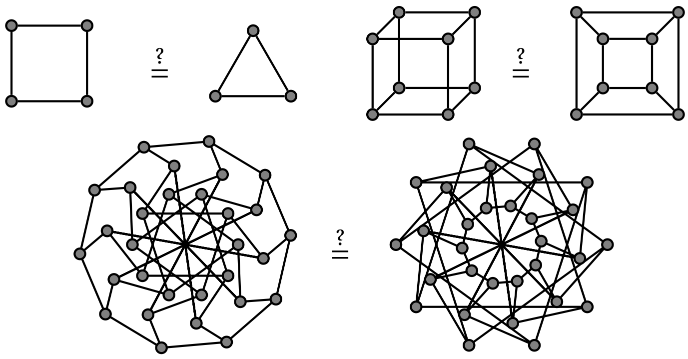


FIGURE 2.3 – Trois instances du problème d'isomorphisme de graphes.

2.6 Et en pratique ?

Comme on l'a dit à plusieurs reprises dans ce chapitre, la complexité d'un algorithme nous permet de le classer dans les choix qu'on va réaliser pour résoudre un problème donné.

Si seul le temps de calcul nous intéresse, et qu'on a le choix entre un algorithme A en $O(2^n)$ et un algorithme B en $O(\log n)$, il n'y a en général pas à hésiter : on choisira B pour résoudre notre problème. Il arrivera cependant qu'on doive choisir entre plusieurs algorithmes de même complexité (par exemple en $O(n)$), et il faudra alors pousser l'analyse plus loin pour déterminer quel algorithme choisir. On en verra quelques exemples dans les exercices ; en voici déjà un.

6. La troisième instance vient d'ici : `http://texample.net/tikz/examples/combinatorial-graphs/`.

Exemple 11 (même complexité, temps d'exécution différents)**.** Les deux fonctions ci-dessous sont en $O(n)$:

```
def fonction_1(n):
    for i in range(n):
        pass

def fonction_2(n):
    for i in range(n):
        print("Bonjour!")
```

`fonction_1` ne faisant rien, elle s'exécutera nécessairement plus vite que `fonction_2`. En pratique, on obtient par exemple 0,001735 secondes pour `fonction_1` et 0,01268 secondes pour `fonction_2` pour $n = 100\,000$.

De plus, il est en général préférable d'utiliser les fonctions déjà disponibles dans un langage plutôt que de les réécrire soi-même : non seulement il est inutile de réinventer la roue, mais ces fonctions sont en outre robustes : elles ont déjà été testées par des millions d'utilisateurs pendant plusieurs décennies sur d'innombrables jeux de tests pour la plupart d'entre elles, et elles sont bien souvent plus performantes que ce que de nouveaux venus pourraient écrire. Ceci est particulièrement vrai en Python, car les fonctions de la bibliothèque standard ont été écrites en C et sont compilées, ce qui fait que les fonctions que nous écrirons, qui sont elles interprétées plutôt que compilées, pourront rarement rivaliser avec celles des développeurs de Python, même si la complexité est la même. Dans certains cas surprenants, il arrivera même que nos fonctions aient une meilleure complexité mais soient plus lentes ! Il s'agit cependant là d'une particularité de Python qui n'existe pas forcément dans les autres langages.

On peut se demander en pratique quelles complexités sont acceptables. Une réponse expéditive consisterait à rejeter automatiquement les algorithmes en $O(n^3)$, par exemple ; mais il est nécessaire d'avoir plus de contexte pour répondre à cette question. Dans certaines situations, il est possible qu'on ne puisse pas faire mieux que $O(n^3)$; dans d'autres situations, les données pourraient être tellement gigantesques que même un algorithme en $O(n)$ serait trop lent.

2.7 Remarques

2.7.1 Le module `timeit`

Le lecteur curieux pourra consulter la documentation du module `timeit`, qui est dédié à la mesure du temps de calcul de morceaux de code en Python et réalise donc les tâches que l'on a accomplies ici avec les fonctions du module `time`. On a évité d'en parler dans ce chapitre pour deux raisons :

1. comme précisé dans l'introduction, notre but est de couvrir des techniques plus générales que ce qui existe en Python ; `timeit` étant un module spécifique à Python, on ne peut donc pas compter sur sa présence dans d'autres langages. Par contre, la plupart des langages possèdent des analogues de `time()` ou `clock()`, et l'on pourra

donc facilement mesurer le temps d'exécution du code qui nous intéresse de la même manière dans ces autres langages ;

2. la syntaxe à utiliser pour utiliser le module `timeit` peut se révéler plus subtile quand on a besoin d'utiliser des fonctions importées d'autres modules, au contraire de l'approche présentée ici qui fonctionne toujours de la même manière.

2.7.2 Complexité des opérations élémentaires

On a supposé ici que toutes les opérations basiques étaient en $O(1)$. En réalité, il est essentiel de disposer d'algorithmes efficaces pour les réaliser : les méthodes apprises à l'école pour additionner, multiplier, diviser et soustraire deux entiers sur n chiffres se révéleront trop lentes — $O(n)$ pour l'addition et la soustraction, et $O(n^2)$ pour la multiplication et la division. De nombreux chercheurs ont réussi à obtenir de meilleurs algorithmes ; parmi les découvertes récentes, citons l'algorithme de Harvey et van der Hoeven [1] qui permet de multiplier deux nombres sur n chiffres en $O(n \log n \, 4^{\log^* n})$. Les mêmes auteurs ont récemment soumis un manuscrit décrivant un meilleur algorithme en $O(n \log n)$.

Bibliographie

[1] D. Harvey et J. van der Hoeven, *Faster integer multiplication using plain vanilla FFT primes*, Mathematics of Computation, 88 (2019), pages 501–514.

[2] S. Skiena, *The Algorithm Design Manual*, Springer, 2ème édition, 2008.

Chapitre 3

Algorithmes sur les séquences

Sommaire

Tout au long de ce chapitre, on utilisera le terme **séquence** pour désigner un itérable ordonné dont les éléments sont accessibles à l'aide d'un indice en temps $O(1)$. Ce terme générique nous permettra de traiter de la même façon les listes et les tuples en Python, les tableaux en C/C++, etc. On fera également l'hypothèse suivante :

Les séquences qu'on manipule ne contiennent que des nombres entiers.

Les algorithmes qu'on y étudiera fonctionneront parfaitement sans la moindre modification pour les autres types, mais cette hypothèse nous permettra de simplifier grandement leur analyse, car on pourra affirmer que comparer deux éléments de la séquence coûte un temps constant, alors que le temps nécessaire à la comparaison de deux chaînes de caractères par exemple serait proportionnel à la longueur de ces chaînes.

3.1 Les bases

3.1.1 Complexité des opérations sur les listes en Python

Exercice 2. Si `L` est une liste de n entiers, `M` est une liste de m entiers, et si `x` et `i` sont entiers, quelle est la complexité des opérations suivantes et pourquoi ?

— `L.append(x)` :

— `L.extend(M)` :

— `L.index(x)` :

— `L.insert(i, x)` :
— `L.pop()` :
— `L.pop(i)` :
— `L.remove(x)` :

En pratique, la complexité dépend de ce qu'on insère et dans quoi on l'insère; par exemple, si l'on fait un `L.extend(M)` d'une liste d'entiers, la complexité dépendra de la taille de `M`.

3.1.2 Listes et fonctions en Python

⚠ Attention : en Python, les listes sont des types modifiables et se comportent donc différemment des types non modifiables quand on les passe en paramètre à des fonctions!

Exemple 12 (listes comme paramètres de fonctions)**.** Voici une comparaison de l'effet de deux fonctions sur leurs paramètres en fonction de leur type :

```
>>> def truc(a):
...     a = a + 1
...
>>> a = 5
>>> truc(a)
>>> a
5
>>> def truc_liste(L):
...     L[0] = L[0] + 1
...
>>> L = [1, 2, 3]
>>> truc_liste(L)
>>> L
[2, 2, 3]
```

Les fonctions modifient les listes passées en paramètres, alors que ce n'est pas le cas des autres variables. Il faut donc être prudent quand on passe une liste en paramètre; l'avantage est qu'il n'est pas nécessaire de renvoyer un résultat modifié.

3.2 Recherche dans une séquence

La recherche d'un élément dans une structure est une tâche extrêmement fréquente (il suffit de penser aux bases de données). On verra ici deux algorithmes nous permettant de trouver une occurrence d'un élément dans une séquence et de renvoyer sa position ou `None` le cas échéant : la recherche linéaire et la recherche dichotomique.

On fera les hypothèses de complexité suivantes :

1. créer une séquence `T` de n entiers coûte $O(n)$;
2. accéder à l'élément `T[i]` coûte $O(1)$;
3. appeler la fonction `len()` coûte $O(1)$.

Le problème qu'on cherche à résoudre est le suivant :

RECHERCHE D'UN ÉLÉMENT
Données: une séquence `T`, un élément `x`.
Question: est-ce que `T` contient `x` ? (si oui, donner sa position)

3.2.1 Recherche linéaire

Une manière simple pour trouver un élément dans une liste est d'effectuer un parcours en comparant chaque élément rencontré à celui qu'on cherche ; en cas d'égalité, on renvoie la position de l'élément examiné. L'algorithme 1 implémente cette stratégie.

```
def recherche_lineaire(T, x):
    """Renvoie la position de x dans T, ou None si T ne contient pas x.
    Fonctionne sur n'importe quel itérable indiçable par un intervalle de
    naturels (liste, tuple, chaîne)."""
    for i in range(len(T)):  # examiner chaque position
        if T[i] == x:        # on a trouvé x
            return i
```

ALGORITHME 1: Recherche linéaire.

Remarquons qu'il s'agit de l'algorithme utilisé par Python lorsqu'on fait appel à `if x in T` ou `T.index(x)`. Attention, `in` ne renvoie pas de position, et `index` plante si `T` ne contient pas `x` ! Si l'on tient à se restreindre à ce qui est déjà implémenté en Python, cela nous oblige donc à faire deux parcours de `T` : un premier parcours pour vérifier si `x` est dans `T`, et un second parcours pour obtenir sa position le cas échéant[1].

Exercice 3. Adaptez l'algorithme de recherche linéaire pour qu'il renvoie *toutes* les positions de `T` contenant `x`.

Correction

L'algorithme est correct, puisqu'on passe bien en revue toutes les positions de l'itérable jusqu'à ce qu'on trouve ce qu'on cherche ou qu'on ait la garantie que l'itérable examiné ne contient pas l'élément cherché.

Complexité

La recherche linéaire porte bien son nom : toutes les opérations qu'elle effectue sont en temps constant, et l'on est obligé dans le pire cas de parcourir toute la liste pour se rendre compte qu'elle ne contient pas l'élément. Dès lors, elle est en $O(n)$. Peut-on prouver que c'est optimal, ou qu'il existe mieux ?

3.2.2 Recherche dichotomique

Si l'on ne sait rien d'autre sur la séquence examinée, on ne peut pas faire mieux que la recherche linéaire. En effet, dans le pire des cas, on est obligé de lire tous les éléments pour

1. En pratique, on aura plutôt recours au mécanisme des *exceptions* pour éviter le premier parcours.

être sûr que l'élément recherché n'est pas dans la liste. Si en revanche `T` est triée, on peut aller beaucoup plus vite en procédant par **dichotomie**, qui signifie littéralement “couper en deux”.

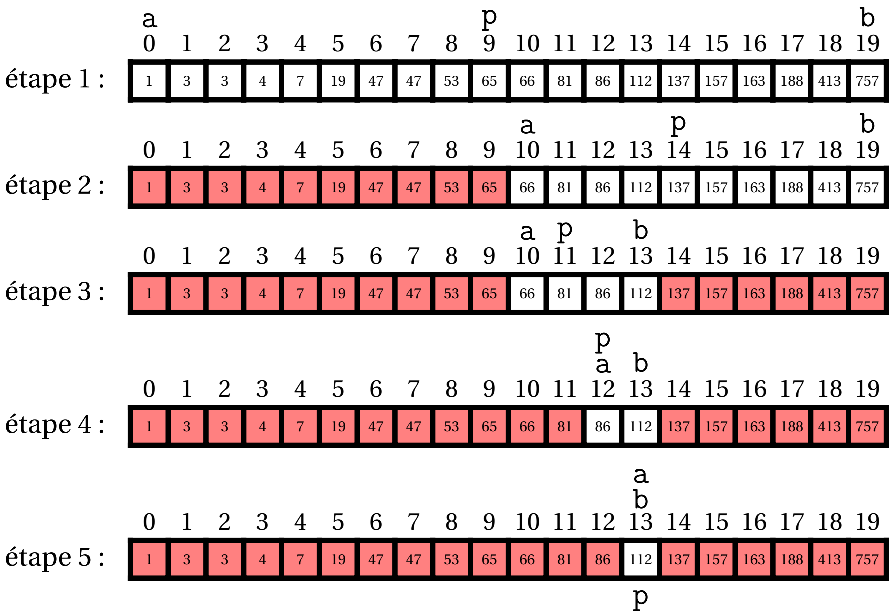
Il s'agit du même principe que l'on utilise pour chercher un mot dans un dictionnaire papier, qui est forcément trié : au lieu de lire tous les mots les uns à la suite des autres, on ouvre le livre à peu près au milieu, et en fonction de la page sur laquelle on tombe, on devra soit poursuivre notre recherche de la même manière dans la première moitié ou dans la seconde moitié, soit s'arrêter car on a trouvé la page contenant l'information qu'il nous faut.

Dans notre cas plus simple basé sur les listes, on peut imiter ce processus en le décrivant comme suit :

1. on examine l'élément du milieu, en position `p` ;
2. si `T[p]` vaut `x`, alors on a fini ;
3. sinon, on élimine la moitié de la liste et on poursuit notre recherche sur le reste selon le même principe.

Avant de présenter l'implémentation de l'algorithme, illustrons son action sur un exemple simple.

Exemple 13. Utilisons la dichotomie pour rechercher l'élément 112 dans la liste ci-dessous. Les zones rouges indiquent les éléments ignorés :



L'algorithme 2 montre une implémentation où les variables `a`, `b` et `p` ont reçu les noms plus parlants `debut`, `fin` et `milieu`. On suppose que les éléments que la liste contient sont comparables à `x` — sinon le code risque de planter : l'opérateur == fonctionnera toujours en Python, mais ceux qui demandent quel élément est inférieur ou supérieur à un autre requièrent que les types soient comparables.

```python
def recherche_dichotomique(T, x):
    """Renvoie la position de x dans T, ou None si T ne contient pas x.
    Fonctionne sur n'importe quel itérable indiçable par un intervalle de
    naturels (liste, tuple, chaîne)."""
    debut = 0
    fin = len(T) - 1
    while debut <= fin:                # recherche entre les positions debut et fin
        milieu = (debut + fin) // 2   # examiner l'élément du milieu
        if T[milieu] == x:            # on a trouvé x
            return milieu
        if T[milieu] > x:             # éliminer les "supérieurs"
            fin = milieu - 1
        else:                         # éliminer les "inférieurs"
            debut = milieu + 1
```

ALGORITHME 2: Recherche dichotomique.

Exercice 4. L'algorithme 2 présenté ci-dessus fonctionne dans le cas où la séquence est triée de manière croissante. Donnez une version adaptée au cas où la séquence est triée de manière *décroissante*.

Correction

Pour se convaincre que la dichotomie fonctionne, il faut s'assurer que les choix éliminés à chaque étape sont corrects, c'est-à-dire qu'on ne s'interdit jamais d'aller visiter des emplacements qui pourraient contenir la valeur cherchée. L'hypothèse selon laquelle la séquence examinée est triée est bien entendu cruciale; en effet, lorsqu'on sélectionne l'élément e en position p, la liste se présente comme suit :

0	p	$n-1$
$\leq e$	e	$\geq e$

Dès lors :

1. si e est plus grand que l'élément recherché, c'est forcément aussi le cas de tous les éléments d'indice $\geq p$ puisque la liste est triée; il est donc correct d'ignorer tous les éléments d'indice $\geq p$ dans la suite de la recherche.
2. si e est plus petit que l'élément recherché, c'est forcément aussi le cas de tous les éléments d'indice $\leq p$ puisque la liste est triée; il est donc correct d'ignorer tous les éléments d'indice $\leq p$ dans la suite de la recherche.

On est également certain que l'algorithme finira par se terminer, puisqu'à chaque étape infructueuse, soit l'indice de début augmente, soit l'indice de fin décroît, ce qui garantit que la condition d'arrêt `debut > fin` finira par être vérifiée.

Complexité

Toutes les opérations effectuées dans l'algorithme 2 sont en $O(1)$; pour connaître sa complexité, il nous suffit donc de calculer le nombre d'itérations effectuées dans la boucle principale. Contrairement aux autres algorithmes vus jusqu'ici, on ne voit pas directement quel est ce nombre. Pour le calculer, on aura besoin de la notion de *logarithme*.

Définition 5. Le **logarithme en base** b de x, noté $\log_b x$, est le nombre ℓ tel que $x = b^\ell$.

Les propriétés suivantes des logarithmes seront également utiles :

1. $\log_b b^x = x$, car par définition $\log_b b^x = m$ tel que $b^x = b^m$;
2. $x = y \Leftrightarrow \log_b x = \log_b y$.

Pour calculer la complexité de la recherche dichotomique, il nous faut maintenant déterminer le pire cas, ainsi que le nombre d'itérations qu'on y effectue. Comme pour la recherche linéaire, le pire cas est celui où la liste ne contient pas l'élément que l'on cherche, car on est alors obligé d'effectuer le maximum d'itérations de la boucle pour s'en rendre compte. La recherche s'arrête donc au pire après avoir échoué sur un intervalle de taille 1, et on va donc calculer le nombre c d'itérations nécessaires pour obtenir un intervalle de cette taille. Comme chaque itération divise notre intervalle de recherche de n éléments par 2, on doit résoudre l'équation suivante :

$$\frac{n}{\underbrace{2 \times 2 \times \cdots \times 2}_{c \text{ fois}}} = 1.$$

Autrement dit :

$$2^c = n \Leftrightarrow \log_2 2^c = \log_2 n \Leftrightarrow c = \log_2 n.$$

La recherche dichotomique est donc en $O(\log n)$ (on verra un peu plus bas pourquoi on n'écrit pas $O(\log_2 n)$), ce qui comme le montre la Figure 3.1 est bien plus rapide que la recherche linéaire.

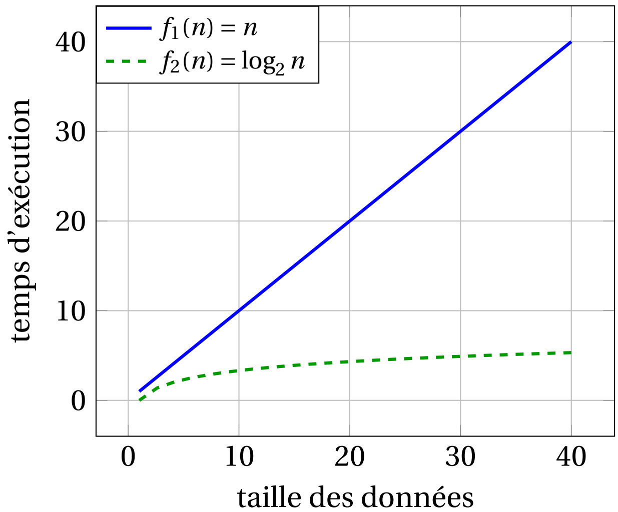


FIGURE 3.1 – Croissance logarithmique contre croissance linéaire.

Rappelons que l'on peut convertir un logarithme en une base vers un logarithme en une autre base à l'aide de la formule suivante : $\log_b n = \frac{\log_c n}{\log_c b}$. Ces fonctions diffèrent par leurs bases mais sont équivalentes **du point de vue de** $O(\cdot)$. La Figure 3.2 illustre cette croissance logarithmique.

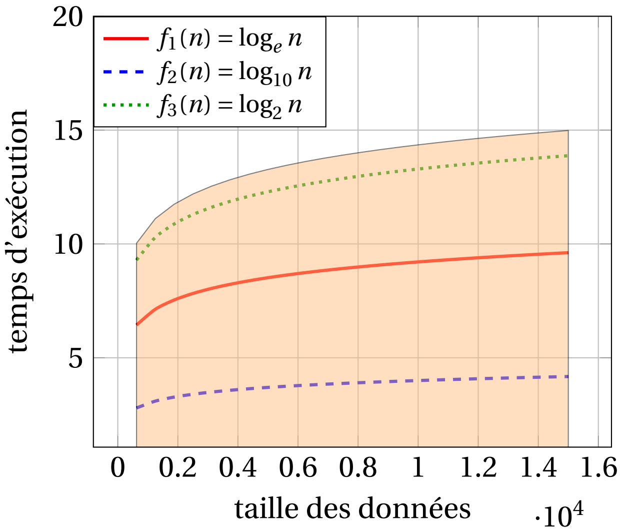


FIGURE 3.2 – Croissance logarithmique.

On écrira donc $O(\log n)$ au lieu de $O(\log_2 n)$, $O(\log_{10} n)$, ... puisqu'on peut appliquer la règle vue en sous-section 2.3.1 qui nous permet d'éliminer les constantes multiplicatives.

On pourrait également chercher des éléments sur base d'autres critères ; par exemple, on peut vouloir trouver le plus petit ou le plus grand élément d'une liste. Là encore, l'hypothèse selon laquelle la séquence est triée de manière croissante aide : le minimum est en première position, et le maximum en dernière position. On peut donc trouver ces éléments en $O(1)$ si `T` est triée (contre $O(n)$ dans le cas général).

Exercice 5. Supposons maintenant qu'on cherche *toutes* les occurrences d'un élément dans une séquence. Quelle complexité peut-on obtenir ? Peut-on améliorer ce résultat si la séquence est triée ?

3.3 Tri de listes

Comme on l'a vu avec la recherche d'éléments, il est important pour des raisons d'efficacité que les structures sur lesquelles on travaille soient organisées d'une certaine manière. On va donc maintenant voir comment **trier** des listes. On veut donc résoudre le problème suivant :

TRI	
Données:	une liste `T`.
Objectif:	ordonner les éléments de `T` de manière croissante.

La seule hypothèse cruciale que l'on fera pour que les algorithmes examinés fonctionnent est que les éléments de notre liste sont comparables.

Trois algorithmes basiques de tri sont généralement examinés en guise d'introduction au problème de tri : le tri bulle, le tri par sélection, et le tri par insertion. Le tri bulle étant toujours le moins efficace des trois, on se contentera ici de présenter le tri par sélection et le tri par insertion.

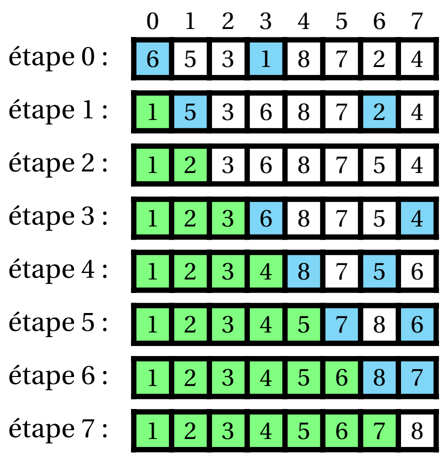
3.3.1 Le tri par sélection

Le *tri par sélection* consiste à parcourir la liste, en échangeant à chaque étape l'élément que l'on est en train de lire avec le plus petit parmi ceux qu'il nous reste à lire. Ainsi, après la première étape, le minimum de la liste est en première position ; après la seconde étape, le deuxième plus petit élément de la liste est en deuxième position, et ainsi de suite. Plus formellement :

1. on examine chaque élément à tour de rôle en partant du premier ;
2. à l'étape `i`, on échange `T[i]` avec l'élément le plus petit entre les positions `i` et `n-1`.

Lorsque l'algorithme se termine, la liste est bien triée. En effet, après l'étape `i`, on a dans les `i` premières positions de `T` les `i` plus petits éléments de `T` dans le bon ordre ;

Exemple 14. Voici les étapes effectuées par le tri par sélection sur la liste `[6, 5, 3, 1, 8, 7, 2, 4]` ; les éléments qui vont subir un échange sont en bleu, ceux qui ne bougeront plus sont en vert.



Pour implémenter cet algorithme, il faut donc être capable de trouver la **position** du minimum d'une liste entre deux positions données, puisque les positions sont nécessaires pour modifier la liste donnée. Il est important de pouvoir se limiter à un sous-intervalle précis de la liste à trier, puisqu'on ne désire plus déplacer les éléments déjà triés. L'algorithme 3 montre une fonction réalisant cette tâche (`T` est supposée non vide, `i` et `j` valides), qui s'obtient très facilement à partir de l'algorithme classique de recherche du minimum.

Maintenant qu'on sait comment trouver la position du minimum, on peut écrire l'algorithme 4, qui implémente le tri par sélection en suivant exactement la description que l'on a donnée un peu plus haut.

Comment prouver que l'algorithme est bien correct ? On peut procéder par induction sur la longueur d'un préfixe déjà trié. Le cas de base est trivial : si la longueur du préfixe vaut 0, alors ce préfixe est bien trié. Pour l'induction, supposons que l'algorithme a déjà trié les

```python
def position_minimum(T, i, j):
    """Renvoie la position du plus petit élément de la liste T entre les
    positions i et j comprises."""
    position = i
    for p in range(i + 1, j + 1):
        if T[p] < T[position]:  # on a trouvé plus petit
            position = p        # mettre à jour la position
    return position
```

ALGORITHME 3: Recherche de la position du minimum d'une liste dans un intervalle donné.

```python
def tri_selection(T):
    """Trie la liste donnée en entrée selon l'algorithme du tri par sélection."""
    n = len(T) - 1
    # pour chaque position de la liste sauf la dernière
    for i in range(n):
        p = position_minimum(T, i, n)   # trouver le minimum de T[i], ..., T[n]
        T[p], T[i] = T[i], T[p]         # échanger ce minimum avec T[i]
```

ALGORITHME 4: Tri par sélection.

k premières entrées ; cela veut dire qu'en première position, on a le minimum de toute la liste, en deuxième position, on a le deuxième plus petit élément de la liste, ... et en $k^{\text{ème}}$ position, on a le $k^{\text{ème}}$ plus petit élément de la liste. On est donc en train d'examiner l'élément en position $k+1$, et le tri par sélection va l'échanger avec le $(k+1)^{\text{ème}}$ plus petit élément de la liste, qui se trouve forcément après la position k. Ainsi, chaque étape du tri permet de passer d'une solution partielle correcte de taille k à une solution partielle correcte de taille $k+1$, et l'algorithme se terminant après avoir lu toute la liste, elle est bien triée à la fin de l'exécution de l'algorithme.

Exercice 6. Quelle est la complexité du tri par sélection ?

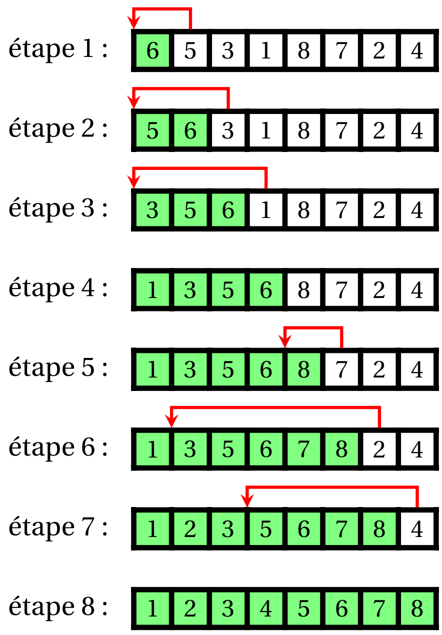
3.3.2 Le tri par insertion

Une autre approche du tri consiste, lors du parcours, à déplacer l'élément actuel de manière à ce qu'il se retrouve en bonne place par rapport à ce que l'on a déjà trié. C'est ce que réalise le *tri par insertion*, qui s'économise donc une recherche du minimum et réinsère l'élément en cours de lecture au lieu de simplement l'échanger avec un autre. Plus formellement :

1. on examine chaque élément à tour de rôle en partant du deuxième ;
2. à l'étape `i`, on insère `T[i]` "à la bonne place" parmi les `i` premiers éléments.

Lorsque l'algorithme se termine, la liste est bien triée. En effet, après l'étape `i`, les éléments de `T[0]` à `T[i]` sont triés.

Exemple 15. Voici les étapes effectuées par le tri par insertion sur la liste `[6, 5, 3, 1, 8, 7, 2, 4]` :

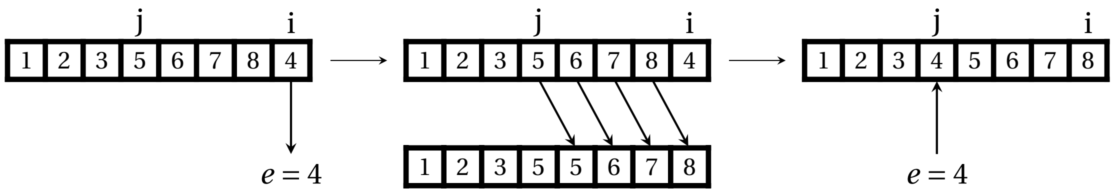


- *(fin exemple 15)* - -

Pour mettre en œuvre le tri par insertion, il faut donc savoir comment effectuer les réinsertions. On peut procéder en trois étapes :

1. sauvegarder l'élément `e` à déplacer;
2. décaler les éléments d'une position de manière à créer une place libre à la destination;
3. affecter la valeur de `e` à la destination;

Exemple 16. Voici les trois étapes de l'algorithme de réinsertion sur `[1, 2, 3, 5, 6, 7, 8, 4]`, avec $i = 7$ et $j = 4$:



L'algorithme 5 implémente cette approche. Une fois cette fonction implémentée, l'implémentation de l'algorithme du tri par insertion est simple : il ne nous reste plus qu'à effectuer un parcours, dans lequel on cherche, pour chaque élément, la position à laquelle il faut le réinsérer dans le préfixe déjà trié; pour la réinsertion proprement dite, on fait appel à l'algorithme 5. L'algorithme 6 montre le tri complet.

Exercice 7. Quelle est la complexité du tri par insertion ?

Exercice 8. Prouvez que le tri par insertion est correct. *(Aide : vous pouvez suivre la stratégie utilisée pour le tri par sélection)*

```python
def reinsertion(T, i, j):
    """Réinsère T[i] en position j. On suppose j <= i valides."""
    # sauvegarder l'élément à déplacer
    e = T[i]
    # remonter d'une position les éléments entre i et j
    for k in range(i, j, -1):
        T[k] = T[k-1]
    # copier l'élément à déplacer en position j
    T[j] = e
```

ALGORITHME 5: Réinsertion d'un élément avant un autre dans une liste.

```python
def tri_insertion(T):
    """Trie la liste donnée en entrée selon l'algorithme du tri par insertion."""
    # pour chaque position de la liste
    for i in range(1, len(T)):
        # chercher à quelle position réinsérer T[i]
        for j in range(i-1, -1, -1):
            if T[j] <= T[i]:
                break
        # effectuer la réinsertion
        reinsertion(T, i, j + 1)
```

ALGORITHME 6: Tri par insertion.

3.4 Le tri pair / impair

Dans certaines situations, il n'est pas nécessaire de trier entièrement les données qui sont disponibles : on veut se contenter de grouper les éléments par catégories. Cela pourrait être le cas par exemple d'un système dans lequel les tarifs imposés aux clients varient en fonction de leur âge ; dans ce cas-là, il est important de distinguer les clients mineurs des clients agés, mais il n'est pas nécessaire de pousser le tri jusqu'à ordonner les clients par âge croissant ou par nom.

Un exemple simplifié de ce genre de situation est le *tri pair / impair* : étant donnée une séquence de nombres naturels, notre but est de réorganiser cette séquence de manière à ce que les éléments pairs précèdent les éléments impairs dans le résultat. Il n'est pas important que les nombres eux-mêmes soient triés : on veut simplement les séparer en fonction de leur parité.

On ne peut malheureusement pas réutiliser les algorithmes de tri que l'on a déjà vus, puisqu'ils ne séparent pas les éléments pairs des éléments impairs. De plus, comme on est moins exigeant sur le résultat que pour les algorithmes de tri vus jusqu'ici, on se dit qu'il doit être faisable d'arriver plus rapidement à nos fins que si l'on triait complètement les données — autrement dit, qu'il doit exister un algorithme de complexité inférieure à ceux qu'on a déjà vus. Comment procéder ?

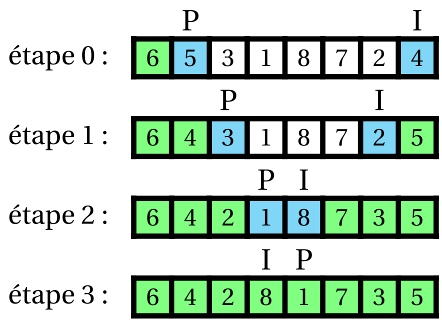
Les quelques idées naïves auxquelles on peut penser nous donneront des algorithmes trop lents. Pour découvrir un algorithme *efficace* qui pourrait fonctionner, il est souvent utile de se demander comment faire pour partir d'une configuration partiellement correcte et lui appliquer une modification qui la rend “plus correcte”. Par exemple, dans le cas des algorithmes de tri classiques, on partait d'une séquence dont les k premiers éléments étaient triés, et on voulait en déduire une séquence dans laquelle les $k+1$ premiers éléments étaient triés. Ici, on pourrait par exemple supposer que les k_1 premiers éléments de la liste sont pairs, les k_2 derniers éléments de la liste sont impairs, et l'on voudrait arriver à une situation où les $k_1 + 1$ premiers éléments sont pairs et les $k_2 + 1$ derniers éléments sont impairs.

Soit P la position de l'élément suivant les k_1 premiers éléments pairs, et I la position précédant les k_2 derniers éléments impairs. Tout ce qui précède P est donc correct, tout ce qui suit I aussi, et l'on doit maintenant s'occuper de la zone entre P et I. On a plusieurs cas à examiner :

1. si `liste[P]` est pair, alors cet élément ne doit pas bouger et l'on peut donc incrémenter P ;
2. de même, si `liste[I]` est impair, alors cet élément ne doit pas bouger et l'on peut donc décrémenter I ;
3. si `liste[P]` est impair, alors cet élément n'est pas dans la bonne zone et il devra donc être déplacé ;
4. de même, si `liste[I]` est pair, alors cet élément n'est pas dans la bonne zone et il devra donc être déplacé.

Pour déplacer les éléments hors de leur zone, on va les échanger. Il faut donc s'assurer que l'échange règle exactement deux problèmes à la fois, pour qu'après cet échange, toutes les éléments jusque à la position P comprise soient pairs et tous les éléments à partir de I compris soient impairs. On va donc faire avancer P tant que `liste[P]` est pair, et descendre I tant que `liste[I]` est impair.

Exemple 17. Voici les étapes effectuées par le tri pair / impair sur la liste `[6, 5, 3, 1, 8, 7, 2, 4]`. Les éléments en vert sont déjà dans la “bonne” zone (celle qui est censée les contenir) ; ceux en bleu vont subir un échange.



L'algorithme 7 implémente le tri pair / impair. La présence d'une boucle `while True` nous empêche de calculer sa complexité en appliquant mécaniquement les règles vues au chapitre 2 ; nous allons donc procéder autrement pour compter le nombre d'opérations :

— le curseur `pair` subit au plus n incrémentations ;

— le curseur `impair` subit au plus n décrémentations ;

— les seules autres opérations que l'on effectue, y compris les échanges, sont toutes en $O(1)$.

Nous avons donc une complexité en $O(n)$.

```python
def tri_pair_impair(liste):
    """Place les éléments pairs d'une liste d'entiers au début et ses éléments
    impairs à la fin."""
    n = len(liste)
    pair = 0
    impair = n - 1
    while True:
        # avancer le curseur pair vers le prochain élément impair
        while pair < n and not liste[pair] % 2:
            pair += 1
        # reculer le curseur impair vers le prochain élément pair
        while impair >= 0 and liste[impair] % 2:
            impair -= 1
        # si les curseurs se croisent, on a fini
        if pair >= impair:
            return
        # sinon, on échange les éléments mal placés
        liste[pair], liste[impair] = liste[impair], liste[pair]
```

ALGORITHME 7: Tri pair / impair.

Exercice 9. Soit `T` une liste de n éléments qui a été triée par le tri pair / impair. Adaptez l'algorithme de recherche dichotomique pour trouver, en $O(\log n)$, la position de la frontière entre les deux catégories d'éléments — c'est-à-dire la position `x` telle que tous les éléments de `T` jusqu'à `T[x]` inclus sont pairs, et tous les éléments à partir de `T[x+1]` sont impairs.

3.5 Remarques

De nombreux autres algorithmes de tri bien plus efficaces que ceux montrés ici existent (voir par exemple Cormen et al. [2], Knuth [3], et d'autres cours plus avancés). Ceux que l'on a choisi de couvrir dans ce chapitre font partie des méthodes les plus basiques ; ils présentent l'avantage d'être simples à expliquer, à comprendre, et donc à implémenter. De plus, certaines de leurs idées nous seront utiles dans d'autres contextes.

Ces algorithmes basiques de tri nous fournissent également un exemple de cas où la notation $O(\cdot)$ élimine un peu trop d'informations pour nous permettre de faire un bon choix. En effet, les algorithmes vus sont de même complexité, et pourtant leurs performances en pratique peuvent différer assez largement comme on le verra dans les TPs. Ceci est dû en partie au fait que deux paramètres importants influent sur leur efficacité : le nombre de comparaisons qu'ils effectuent, ainsi que le nombre d'affectations.

Les algorithmes de tri disponibles varient selon le langage de programmation que l'on a choisi d'utiliser. En Python comme en Java, l'algorithme qui a été choisi n'est pas l'un de ceux que l'on a couverts, mais un algorithme nommé TimSort [4] dont la complexité est en $O(n \log n)$ [1], et qui est donc bien plus efficace que les méthodes présentées dans ce cours. En pratique, il est donc recommandé la plupart du temps d'utiliser ce qui est implémenté dans le langage ; cela ne rend pas nécessairement les algorithmes vus dans ce chapitre complètement inutiles, car il est fréquent que les algorithmes de tri plus performants utilisent des idées présentes dans d'autres algorithmes plus basiques. C'est le cas de TimSort, qui utilise entre autres des idées du tri par insertion et du *tri par fusion* qu'on verra plus loin.

Enfin, remarquons qu'il est possible d'obtenir une version relativement efficace du tri par insertion : d'une part en ayant recours à la dichotomie pour trouver l'endroit où réinsérer l'élément à déplacer, et d'autre part en utilisant les méthodes disponibles en Python pour effectuer cette réinsertion (un analogue de `list.insert` plutôt que ce qu'on a présenté à l'algorithme 5). Ceci nous donne la version concise et plus efficace présentée à l'algorithme 8 (on renvoie le lecteur à la documentation du module standard `bisect` pour plus d'explications) :

```
from bisect import insort_left

def tri_insertion_bisect(T):
    """Trie la liste T à l'aide de l'algorithme du tri par insertion. La
    recherche de l'endroit où faire la réinsertion se fait par dichotomie,
    mais les décalages restent en O(n)."""
    for i in range(1, len(T)):
        insort_left(T, T.pop(i), 0, i)
```

ALGORITHME 8: Version optimisée du tri par insertion.

Notons qu'il reste utile de savoir comment effectuer une réinsertion efficace comme on l'a vu dans l'algorithme 5, car certains langages (entre autres le C et le C++) disposent de séquences dont on peut modifier le contenu mais pas la taille : on ne peut donc pas se servir d'analogues de `list.pop` ou `list.insert` dans ces situations.

Bibliographie

[1] N. AUGER, V. JUGÉ, C. NICAUD, ET C. PIVOTEAU, *On the worst-case complexity of TimSort*, dans Proceedings of the 26th Annual European Symposium on Algorithms (ESA), édité par Y. Azar, H. Bast, et G. Herman, vol. 112 de LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Août 2018, pages 4 :1–4 :13.

[2] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, ET C. STEIN, *Introduction to Algorithms*, MIT Press, 3ème édition, 2009.

[3] D. E. KNUTH, *The Art of Computer Programming, Volume III : Sorting and Searching*, Addison-Wesley, 1973.

[4] T. PETERS, *Timsort description*.

Chapitre 4

Algorithmique du texte

Sommaire

4.1 Recherche de motifs dans un texte

On parle souvent de **motif** pour éviter de sous-entendre que l'on cherche un vrai mot ayant un sens dans un texte. Le motif pourra donc être par exemple “azilud984+nzal”, qui n'a aucun sens, ou contenir des signes de ponctuation comme “mot, ou phrase, à trouver!”. Il n'y a donc aucun besoin — et ce serait même faux — de séparer le texte dans lequel on cherche nos données en mots.

On veut donc résoudre le problème suivant :

RECHERCHE DE MOTIF
Données: une chaîne `P` de taille `k`, une chaîne `T` de taille `n` ≥ `k`;
Question: est-ce que `T` contient `P`? (si oui, renvoyer la position d'un `P[0]` dans `T`)

En guise d'échauffement, demandons-nous ce qui se cache derrière l'opération `s == t`, où `s` et `t` sont deux chaînes de caractères, et tentons d'écrire une fonction qui réalise la même tâche. Il est important de savoir comment comparer deux chaînes de caractères dans le cas où l'on n'a accès qu'à la comparaison de caractères pour deux raisons :

1. contrairement à Python, tous les langages ne permettent pas de détecter directement si deux chaînes `s` et `t` sont égales grâce à l'opérateur `==`;
2. savoir comment cette comparaison s'effectue nous permettra ensuite de développer des algorithmes plus complexes pour d'autres tâches.

Le principe de comparaison de deux chaînes de caractères est assez simple : deux chaînes de longueurs différentes ne peuvent pas avoir le même contenu, et deux chaînes de même longueur ont le même contenu si et seulement si chaque position dans les deux chaînes contient le même caractère. Si les deux chaînes sont de même longueur, il nous suffit donc de les parcourir en parallèle, et de vérifier si chaque position dans les deux chaînes contient bien le même caractère. On en déduit donc assez facilement l'algorithme 9.

```python
def egalite(S, T):
    """Renvoie True si les deux itérables ont le même contenu, False sinon."""
    if len(S) != len(T):  # longueurs différentes => itérables différents
        return False

    # chercher une différence
    for i in range(len(S)):
        if S[i] != T[i]:   # différence trouvée => itérables différents
            return False

    return True
```

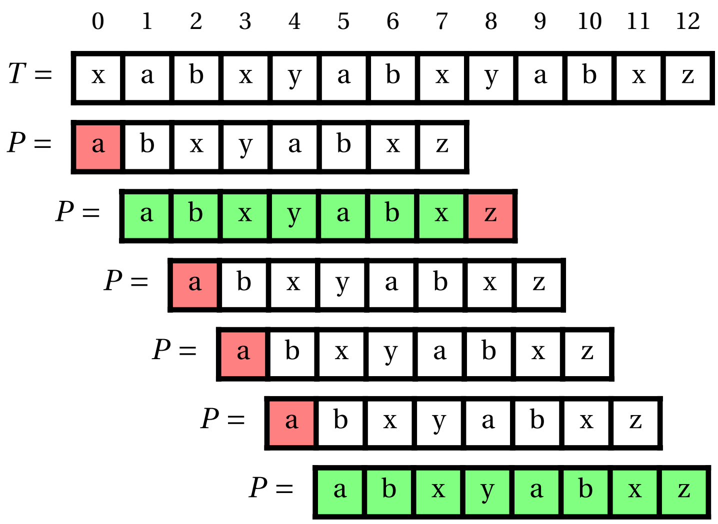
ALGORITHME 9: Comparaison de deux itérables indiçables.

L'idée de l'algorithme "naïf" pour la recherche de motif consiste à essayer toutes les positions valides de T :

1. à chaque position `i`, on teste si
   `(T[i],T[i+1],...,T[i+k-1])` correspond à `(P[0],P[1],...,P[k-1])`
2. si oui : on arrête et on renvoie `i` ;
3. si non : on réessaie avec la position `i+1`.

On s'arrête quand `i > n-k`, puisqu'à partir de cette valeur il n'y a plus assez de place pour contenir `P`.

Exemple 18 (extrait de Gusfield [1]). Supposons que l'on doive trouver le motif "abxyabxz" dans le texte "xabxyabxyabxz" :



L'algorithme 10 renvoie la position de la première occurrence d'un mot dans un texte, ou `None` si le mot n'apparaît pas.

Exercice 10. Il arrive souvent qu'on fasse des fautes de frappe dans un texte. Adaptez l'algorithme 10 pour qu'il admette en plus un paramètre k et renvoie la position d'une séquence dans le texte qui diffère du mot en au plus k positions. Par exemple, si l'on

```python
def recherche(mot, texte):
    """Renvoie la position de la première occurrence du mot dans le texte, ou
    None si texte ne contient pas mot."""
    k = len(mot)
    n = len(texte)
    for i in range(n - k + 1):  # parcourir le texte
        # parcourir le mot
        for j in range(k):
            if mot[j] != texte[i + j]:  # différence -> abandon
                break
        if j == k:                  # on a trouvé
            return i
```

ALGORITHME 10: Recherche naïve d'un mot dans un texte

cherche "bonjour" et si $k = 1$, alors la sous-séquence "binjour" sera acceptée, mais pas la sous-séquence "nonkour".

On teste $n - k + 1$ décalages au total, ce qui correspond donc à $O(n - k)$ itérations de la première boucle. Lors de chaque itération de cette boucle, on parcourt le motif de longueur k, ce qui nous donne $O(k)$ itérations de la seconde boucle dans laquelle on ne fait qu'une opération en temps $O(1)$. Dès lors, la complexité de l'algorithme naïf est $O(k(n - k))$.

Des algorithmes plus sophistiqués permettent d'effectuer la même tâche avec une complexité linéaire en la taille du texte plutôt que quadratique. Ces algorithmes effectuent un prétraitement du motif ou du texte afin de pouvoir procéder à des décalages plus grands en cas d'échec, ce qui nous évite donc de passer en revue toutes les positions du texte. Pour plus d'informations, voir les algorithmes de Knuth-Morris-Pratt et de Boyer-Moore [1].

4.2 Chiffrement

4.2.1 Le chiffre de César

Le chiffre de César consiste à décaler chaque lettre de l'alphabet de k positions ;

Exemple 19. On définit une correspondance entre chaque lettre et sa version chiffrée :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r

Grâce à cette correspondance, on chiffre les messages comme suit :

"message a encoder" ⟶ "ewkksyw s wfugvwj"

Pour chiffrer un caractère `c` :

— il nous faut sa position `p` dans l'alphabet initial ;

— sa position dans l'alphabet modifié est `(p + k) % s` (`s` étant la taille de l'alphabet).

Pour déchiffrer un caractère `d` :

— il nous faut sa position `q` dans l'alphabet modifié ;
— le caractère d'origine est en position `(q - k) % s` dans l'alphabet modifié.

Exercice 11. Écrivez les fonctions permettant de réaliser le chiffrement et le déchiffrement des caractères dans le chiffre de César. Ces fonctions doivent accepter le paramètre k.

En pratique, on peut faire mieux et plus simple avec les bonnes structures (dictionnaires, cf. cours de programmation).

La cryptanalyse s'attache à "casser" les chiffres, c'est-à-dire à décoder les messages chiffrés sans connaître la clé. Dans le cas du chiffre de César, si l'on ne connaît pas la table M, on ne peut en principe pas déchiffrer un message chiffré. Mais en pratique, il n'est pas très difficile de reconstituer M :

1. on sélectionne une portion T de texte chiffré suffisamment grande ;
2. on tente de la déchiffrer en essayant tous les décalages possibles jusqu'à ce que la portion T' déchiffrée ait du sens ;

Il faut donc essayer tous les décalages possibles ... mais comme il n'y en a que 26, c'est vite fait.

4.2.2 Amélioration : chiffrement par substitution

Le décalage d'alphabet utilisé par César est une simple **rotation**. Une rotation n'est qu'une forme particulière de **permutation**, c'est-à-dire une bijection de {'a', 'b', ..., 'z'} vers lui-même. Rien ne nous empêche d'utiliser n'importe quelle autre permutation, ce qui complique grandement la tâche de la personne mal intentionnée voulant déchiffrer le message. C'est le principe du **chiffrement par substitution**, qui décrit n'importe quel chiffrement basé sur une permutation des lettres de l'alphabet utilisé.

Les algorithmes de chiffrement et de déchiffrement ne sont pas très compliqués. Par contre, pour permettre au (à la) destinataire de déchiffrer le message, il faut lui donner *toute la correspondance*, puisque les lettres ne subissent pas toutes le même décalage. L'attaque consistant à passer en revue toutes les possibilités est encore applicable, mais cette fois-ci, le nombre de possibilités est :

$$\begin{aligned} 26! &= 26 \times 25 \times \cdots \times 3 \times 2 \\ &= 403\,291\,461\,126\,605\,635\,584\,000\,000 \\ &= 4{,}032914611 \times 10^{26} \end{aligned}$$

Pour se faire une meilleure idée de ce que représente $4{,}032914611 \times 10^{26}$, rappelons que :

— $1\,000 = 10^3$;
— $1\,000\,000 = 10^6$;
— $1\,000\,000\,000 = 10^9$;

Supposons que notre ordinateur vérifie un million de permutations à la seconde ;

— en une minute, on en vérifie 60×10^6
— en une heure, on en vérifie $60 \times 60 \times 10^6 = 36 \times 10^8$
— en une journée, on en vérifie $24 \times 36 \times 10^8 = 8,64 \times 10^{10}$
— en une semaine, on en vérifie $7 \times 8,64 \times 10^{10} = 6,048 \times 10^{11}$
— en un mois, on en vérifie $4 \times 6,048 \times 10^{11} = 2,4192 \times 10^{12}$
…

Sachant qu'en un an, on vérifie $3,1536 \times 10^{13}$ permutations, on n'aura toujours pas fini après 15 *milliards* d'années.

En pratique, les choses sont plus simples que ça car il existe des techniques plus intelligentes pour réaliser l'attaque telle que l'analyse linguistique et contextuelle, que l'on peut combiner au parallélisme. Des techniques de chiffrement beaucoup plus sûres existent et sont donc plus recommandables.

4.3 Remarques

On a rencontré le terme d'algorithme “naïf” pour la recherche d'un motif dans un texte. C'est un terme qu'on rencontre régulièrement en algorithmique, qui désigne généralement la (ou une des) méthode(s) évidente(s) pour résoudre un problème. Il peut aussi s'agir d'un algorithme appliquant au pied de la lettre une définition pour vérifier qu'un objet qui nous intéresse possède bien une certaine propriété. Bien souvent, les approches naïves présentent l'avantage d'être simples à implémenter, et il vaut mieux avoir un algorithme naïf polynomial que pas d'algorithme du tout. En revanche, ces approches sont rarement efficaces, mais trouver mieux qu'une approche naïve peut se révéler difficile selon le problème auquel on s'attaque.

Python implémente une méthode permettant de résoudre le problème de recherche de motif : `texte.find(motif)` renverra la position de la première occurrence de `motif` dans `texte` — mais seulement si `motif` s'y trouve ; dans le cas contraire, la méthode plante.

Bibliographie

[1] D. GUSFIELD, *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*, Cambridge University Press, 1997.

Chapitre 5

Matrices

Sommaire

Les *matrices* vues au cours de mathématiques apparaissent fréquemment en algorithmique et en programmation, car elles constituent un format très naturel pour stocker les données. Dans un contexte mathématique, ces matrices contiennent des nombres (entiers, réels, complexes, ...). Dans un contexte algorithmique, on peut y mettre à peu près ce qu'on veut — mais si l'on programme un algorithme ayant recours à des matrices, on sera bien sûr limité par ce que le langage nous permet de faire. Elles nous permettent donc de modéliser à la fois des images, des écrans, des animations, des bases de données, des plateaux de jeu, ...

5.1 Les bases

Définition 6. Une **matrice** est un tableau à m lignes et n colonnes ($m, n \in \mathbb{N}$).

Exemple 20. Voici une matrice entière à trois lignes et quatre colonnes :

$$\begin{array}{c|c|c|c|c|}
 & 0 & 1 & 2 & 3 \\ \hline
0 & 3 & 6 & 7 & 1 \\ \hline
1 & 2 & 5 & 6 & 2 \\ \hline
2 & 3 & 1 & 4 & 2 \\ \hline
\end{array}$$

5.1.1 Construction des matrices en Python

Une matrice se représente en Python à l'aide d'une liste de listes.

Exemple 21. Voici comment l'on peut construire et modifier une matrice 3×4 :

```
>>> ligne1 = [3, 6, 7, 1]
>>> ligne2 = [2, 5, 6, 2]
>>> ligne3 = [3, 1, 4, 2]
>>> M = [ligne1, ligne2, ligne3]
>>> M
[[3, 6, 7, 1], [2, 5, 6, 2], [3, 1, 4, 2]]
>>> M[0] = [0, 0, 0, 0]
>>> M
[[0, 0, 0, 0], [2, 5, 6, 2], [3, 1, 4, 2]]
>>> ligne1
[3, 6, 7, 1]
```

- *(fin exemple 21)* - -

On peut également créer une liste de listes vides, qu'on remplira plus tard. **Attention à ne pas se faire avoir :** l'écriture condensée permettant de multiplier une liste de listes vides par un entier n ne donnera pas le résultat voulu ! Au lieu de créer n listes vides différentes, on aura n références à la même liste vide et on s'exposera alors à des problèmes dans la suite du code.

Exemple 22. Si l'on utilise une expression condensée pour initialiser une matrice à n lignes, on ne crée pas n lignes différentes mais bien n copies de la même liste :

```
>>> M = [ [] ] * 10
>>> M[0].append(1)
>>> M
[[1], [1], [1], [1], [1], [1], [1], [1], [1], [1]]
>>> M = [ list() ] * 10
>>> M[0].append(1)
>>> M
[[1], [1], [1], [1], [1], [1], [1], [1], [1], [1]]
```

Au lieu de cette expression condensée, on aura recours soit à une boucle, soit à une **compréhension de listes**.

Exemple 23. Les deux méthodes suivantes fonctionnent pour initialiser une liste de dix listes vides :

```
>>> M = []
>>> for i in range(10):
...     M.append([])
```

ou

```
>>> M = [ list() for i in range(10) ]
```

Dans les deux cas, on peut vérifier que tout marche “correctement” ensuite :

```
>>> M
[[], [], [], [], [], [], [], [], [], []]
>>> M[0].append(1)
>>> M
[[1], [], [], [], [], [], [], [], [], []]  # ouf!
```

Dans la suite du texte, il sera utile de disposer d'une fonction qui initialise et renvoie une matrice avec un nombre spécifique de lignes et de colonnes. Voici comment implémenter une telle fonction :

```
def initialiser_matrice(n, p):
    """Renvoie une matrice de zéros à n lignes et p colonnes."""
    matrice = list()

    for i in range(n):
        matrice.append([0] * p)

    return matrice
```

ALGORITHME 11: Initialisation d'une matrice $n \times p$ de zéros.

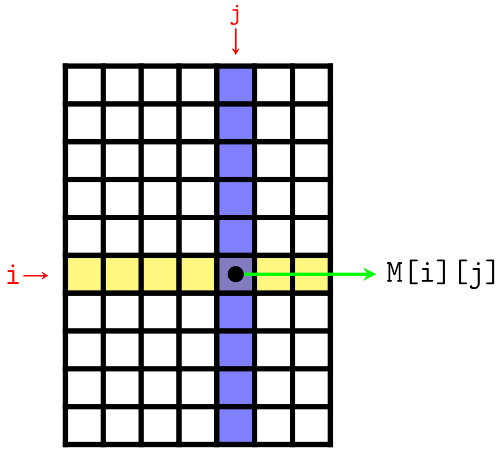
5.1.2 Taille et dimensions

En mathématiques, toutes les lignes d'une matrice ont par définition le même nombre de lignes. En revanche, lorsqu'on programme en Python ou dans un autre langage, rien ne nous force à attribuer le même nombre de colonnes à chaque ligne, ou à nous limiter à deux dimensions. Pour éviter la confusion, on interprétera le terme "matrice" comme en mathématiques (c'est ce qu'on a fait dans la définition 6) ; si l'on veut sous-entendre que toutes les lignes n'ont pas nécessairement la même longueur, on parlera plutôt de "liste de listes".

Comme avant, on peut utiliser la fonction `len(M)` ; mais attention, cette fonction donne le nombre de lignes de `M` ! Ceci est normal car `len(iterable)` renvoie le nombre d'éléments d'un itérable, et les éléments de `M` sont des lignes. Si l'on veut connaître le nombre de cases de `M`, il faut donc additionner les longueurs de toutes les lignes. Si l'on a besoin du nombre de lignes et du nombre de colonnes d'une matrice — non vide ! — `M`, on prendra l'habitude d'écrire : `nb_lignes, nb_colonnes = len(M), len(M[0])`. Cela suffit puisqu'on sait que toutes les lignes ont le même nombre de colonnes. **Attention :** si la matrice est vide, l'instruction `len(M[0])` provoquera une exception ; pour simplifier la présentation, on supposera que nos matrices ne sont jamais vides, mais il faudra faire attention à ce cas particulier en pratique.

5.1.3 Accès aux éléments

Comme avant, on utilise l'opérateur `[]` pour accéder aux éléments d'une matrice. Mais comme on a plusieurs dimensions, il faut spécifier une position par dimension. On précise d'abord le numéro de ligne, et puis le numéro de colonne :



Si l'on indique seulement le numéro de ligne, alors on obtient la ligne entière, donc une liste plutôt qu'un élément. À titre d'exemple, regardons comment créer une matrice dont la ligne `i` contient les multiples de `i+1`.

Exemple 24. Voici comment l'on peut construire une matrice dont la ligne i contient les 10 premiers multiples de $i+1$, en partant de 1 :

```
M = []
for i in range(10):
    M.append(list(range(1, 11)))
    for j in range(10):
        M[i][j] *= (i + 1)
```

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

5.1.4 Recherche d'un élément

Certaines méthodes et certains opérateurs de Python *ne marchent plus pour les matrices*!

Exemple 25. La matrice ci-dessous contient bien les éléments 2 et 3, mais Python ne les trouvera pas :

```
>>> M = [[3, 6, 7, 1], [2, 5, 6, 2], [3, 1, 4, 2]]
```

```
>>> 3 in M
False
>>> M.index(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 2 is not in list
```

--- *(fin exemple 25)* - -

Python ne “fouille” pas les sous-listes, car quand on lui demande de trouver l’élément 2 dans `M`, Python effectue sa recherche parmi les éléments de la liste `M`, qui sont des listes et non des entiers. Il faut donc être capable de se débrouiller sans ça, d’où l’étude des algorithmes que l’on a vus sur les listes (section 3.2).

Adapter les algorithmes vus précédemment aux matrices requiert de prendre en compte les dimensions supplémentaires. Sans grande surprise, il nous faut donc parcourir toute la matrice pour trouver un élément, ce qu’on réalise en parcourant chaque colonne de chaque ligne. L’algorithme 12 illustre la recherche d’un élément dans une matrice. Il s’agit simplement d’appliquer la recherche dans une liste à chaque ligne de la matrice, puisque chaque ligne est une liste.

```python
def recherche_matrice(M, x):
    """Renvoie les coordonnées de l'élément x dans la matrice M, ou None si M ne
    contient pas x."""
    nb_lignes, nb_colonnes = len(M), len(M[0])

    for i in range(nb_lignes):
        for j in range(nb_colonnes):
            if M[i][j] == x:
                return i, j
```

ALGORITHME 12: Recherche d’un élément dans une matrice.

5.2 Produit matriciel

Le produit matriciel s’appuie sur le produit vectoriel, dont la définition est rappelée ci-dessous.

Définition 7. Le **produit de deux vecteurs** $\vec{u}$ et $\vec{v}$ dans $\mathbb{R}^n$ est la somme des produits de leurs composantes :

$$\vec{u} * \vec{v} = \sum_{i=1}^{n} \vec{u}_i \times \vec{v}_i.$$

Exemple 26. Si $\vec{u} = (3, -1, 5.5)$ et $\vec{v} = (-1, 4, 2)$, alors $\vec{u} * \vec{v} = -3 - 4 + 11 = 7$.

On en déduit facilement un algorithme pour calculer cette quantité (voir l’algorithme 13).

```python
def produit(U, V):
    """Renvoie la somme du produit des éléments de deux listes de même longueur;
    les éléments sont supposés compatibles."""
    somme = 0
    for i in range(len(U)):
        somme += U[i] * V[i]
    return somme
```

ALGORITHME 13: Produit de deux vecteurs.

Définition 8. Soit A une matrice $m \times n$ et B une matrice $n \times p$. Notons $A_{i,\rightarrow}$ la $i^{\text{ème}}$ ligne de A et $B_{\downarrow,j}$ la $j^{\text{ème}}$ colonne de B. Le **produit matriciel** $C = AB$ est la matrice $m \times p$ définie par

$$C_{i,j} = A_{i,\rightarrow} * B_{\downarrow,j} \quad \forall 1 \leq i \leq m, 1 \leq j \leq p,$$

où $*$ représente le produit vectoriel.

La Figure 5.1 illustre le produit matriciel et montre pourquoi le nombre de colonnes de A doit être égal au nombre de lignes de B.

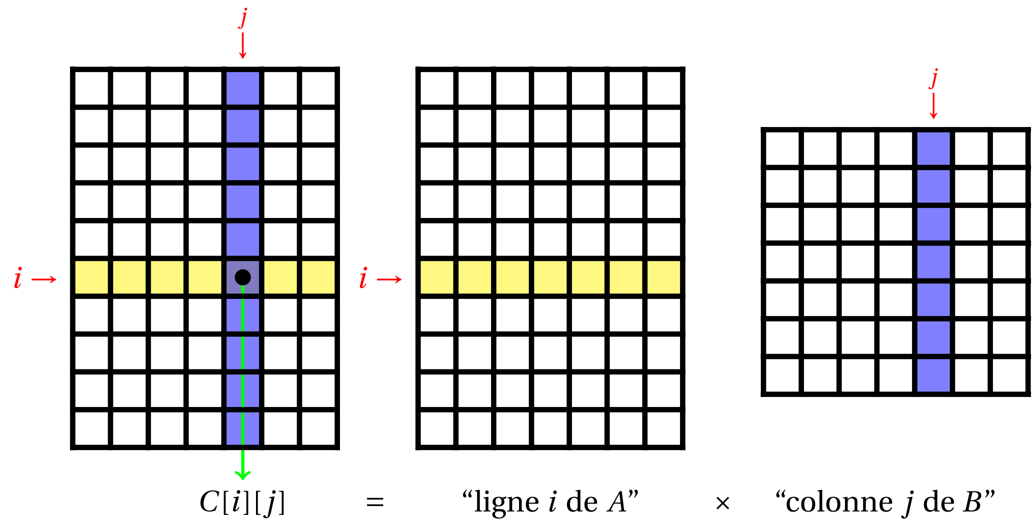


FIGURE 5.1 – Illustration du produit matriciel.

Il est facile de déduire de la définition un algorithme naïf de multiplication matricielle : la $i^{\text{ème}}$ ligne de la matrice A est un vecteur, la $j^{\text{ème}}$ colonne de la matrice B est un vecteur de même taille, et le produit de ces deux vecteurs se retrouve dans la case $C_{i,j}$ du résultat. Le résultat est montré à l'algorithme 14, qui reprend une partie du code de l'algorithme 13 [1].

La complexité de l'algorithme n'est pas difficile à calculer. Supposons que A soit une matrice $m \times n$ et B une matrice $n \times p$. On a trois boucles imbriquées :

1. la première varie sur les lignes de A et comporte m itérations;
2. la deuxième varie sur les colonnes de B et comporte p itérations;

1. Appeler directement cette fonction n'aurait pas été utile, car si on peut accéder facilement aux lignes, on doit recopier les colonnes dans une autre liste avant d'appeler la fonction.

```python
def produit_matriciel(A, B):
    """Renvoie le produit des matrices A et B, dont les types des éléments sont
    supposés compatibles pour le produit, et où le nombre de colonnes de A égale
    le nombre de lignes de B."""
    m, n, p = len(A), len(A[0]), len(B[0])
    C = initialiser_matrice(m, p)
    for i in range(m):
        for j in range(p):
            # produit de la ligne i de A et de la colonne j de B
            somme = 0
            for k in range(n):
                somme += A[i][k] * B[k][j]
            C[i][j] = somme

    return C
```

ALGORITHME 14: Produit matriciel naïf.

3. la troisième parcourt la $i^{\text{ème}}$ ligne de A et la $j^{\text{ème}}$ colonne de B pour en calculer le produit vectoriel, et comporte n itérations.

Ce que l'on fait dans la troisième boucle s'effectue en $O(1)$, et on obtient donc un algorithme en $O(mnp)$. Si les deux matrices sont carrées, c'est-à-dire si $m = n = p$, on a donc un algorithme en $O(n^3)$.

5.3 Remarques

L'algorithme naïf que l'on a vu suit scrupuleusement la définition du produit matriciel. Si les deux matrices à multiplier sont des matrices $n \times n$, le calcul du produit se fait à l'aide de cet algorithme en $O(n^3)$.

Est-il possible de faire mieux? Très étrangement, oui! En 1969, Strassen [1] proposa un algorithme en $O(n^{2.808})$ pour multiplier deux matrices $n \times n$. D'autres algorithmes améliorés furent ensuite découverts, qui rabaissèrent progressivement l'exposant jusqu'au meilleur résultat connu à ce jour : l'algorithme de Williams [2], qui effectue la multiplication en $O(n^{2.3727})$ (voir la Figure 5.2 pour une comparaison graphique des complexités).

Ces chutes progressives de la valeur de l'exposant nous mènent à une autre question ouverte importante en informatique théorique :

Peut-on multiplier deux matrices $n \times n$ en temps $O(n^2)$?

On sait en tout cas qu'il n'est pas possible de faire mieux que $O(n^2)$, car le résultat de la multiplication possède n^2 cases et qu'on a au moins besoin d'un temps $O(1)$ pour écrire le résultat dans chacune de ces cases.

L'intérêt du produit matriciel est qu'il s'agit d'une opération fréquemment utilisée sur les matrices, et beaucoup de problèmes peuvent s'y ramener :

— la factorisation LU dont on parlera plus loin ;

— l'inversion de matrice ;

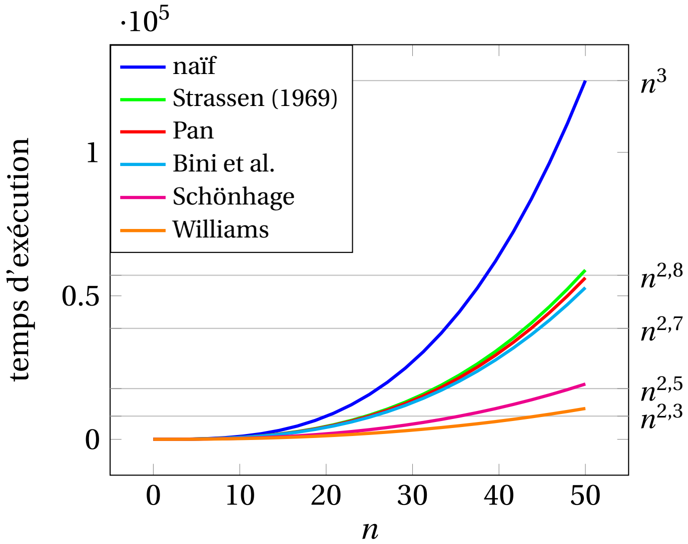


FIGURE 5.2 – Comparaison des complexités de différents algorithmes de multiplication matricielle, pour deux matrices $n \times n$.

— la transformée de Fourier et DCT (utilisées entre autres pour la compression JPEG) ;

— le calcul de plus court**s** chemin**s** dans un graphe, utile pour le calcul d'itinéraires des systèmes GPS ;

$\vdots$

Autrement dit, une solution plus efficace pour le produit matriciel donne automatiquement une solution plus efficace pour les problèmes qui lui sont équivalents (cf. chapitre 2).

Bibliographie

[1] V. STRASSEN, *Gaussian elimination is not optimal*, Numerische Mathematik, 13 (1969), pages 354–356.

[2] V. V. WILLIAMS, *Multiplying matrices faster than Coppersmith-Winograd*, dans Proceedings of the 44th Symposium on Theory of Computing Conference (STOC), édité par H. J. Karloff et T. Pitassi, New York, NY, USA, Mai 2012, ACM, pages 887–898.

Chapitre 6

Compromis calculs / mémoire

Sommaire

Nous avons jusqu'ici focalisé nos critères de performances pour la sélection d'un algorithme sur la complexité. Cependant, les algorithmes ne consomment pas seulement du temps de calcul : ils consomment également de la mémoire, et tout comme le temps de calcul, cette consommation peut se révéler de plus en plus problématique en fonction de la croissance des données sur lesquelles on exécute l'algorithme. Dès lors, ce critère devrait également influer en pratique sur notre choix d'un algorithme.

D'autre part, on peut se rendre compte qu'il existe en pratique un compromis qu'on semble obligé de respecter dans l'implémentation d'un algorithme : il est souvent possible d'accélérer un algorithme en lui permettant d'utiliser plus de mémoire (raison pour laquelle de nombreux algorithmes effectuent un pré-traitement des données avant de commencer leurs tâches), ou au contraire de réduire sa consommation en mémoire, ce qui implique alors un temps de calcul plus important pour compenser ce qu'on ne stocke pas. Remarquons qu'en pratique, des compromis entre d'autres critères peuvent surgir :

— dans le cadre de la compression d'images, on doit faire un compromis entre la taille de l'image et sa qualité : plus l'image est compressée et plus la qualité chute ;

— dans le cadre de la compression de vidéos, si l'espace dont on dispose est fixé, on est souvent amené à choisir entre privilégier le son ou l'image ;

— dans le cadre de la compression de fichiers quelconques, on aura souvent envie de choisir l'algorithme qui réduit le plus la taille des fichiers, mais bien souvent, cela implique un temps de calcul plus important, que ce soit lors de la compression ou lors de la décompression. En pratique, c'est l'utilisation qu'on fera des archives résultantes qui importera : si l'on accède souvent à des fichiers d'une archive, on voudra que cet accès soit rapide, et on se retrouvera donc à devoir faire un compromis entre la taille de l'archive et le temps d'accès.

Nous examinerons dans ce chapitre la problématique de la consommation en mémoire, et quelques techniques illustrant ces compromis entre temps de calcul et consommation en mémoire, avec leurs avantages et leurs inconvénients.

6.1 Complexité spatiale

La rapidité d'un algorithme a déjà été traitée à l'aide de la notation $O(\cdot)$. Nous utiliserons exactement la même notation pour évaluer la consommation en mémoire d'un algorithme, que nous qualifierons de **complexité spatiale**. Attention, on ne mesurera que l'espace *supplémentaire* par rapport aux données en entrée. En effet, si l'on est obligé de travailler sur une liste de n éléments, on ne pourra pas se débarrasser de cette liste : le mieux qu'on puisse faire est donc de limiter la mémoire consommée en plus de cette liste.

Dans le cadre de la complexité algorithmique, on a vu que chaque opération avait un coût en termes de temps de calcul. De la même manière, les variables que l'on utilise ont un coût en mémoire, représenté par des unités (abstraites pour des raisons analogues à ce qu'on a vu avant) :

— une variable “simple” (entier, réel, booléen) a un coût constant $O(1)$;

— une liste de longueur n coûte $O(n)$ unités ;

— une matrice à ℓ lignes et c colonnes coûte $O(\ell c)$ unités ;

$\vdots$

Les règles de calcul déjà vues s'appliquent aussi au calcul de la complexité spatiale. Par exemple, une liste de n “blablas” coûte $O(nt)$ en mémoire, où t est la taille d'un “blabla”.

On a déjà vu comment faire un choix entre deux algorithmes A_1 et A_2 pour résoudre un problème P sur base de leur complexité : on calcule leurs complexités respectives $f_1(n)$ et $f_2(n)$, et si une des complexités est clairement inférieure à l'autre (par exemple : du $O(\log n)$ contre du $O(n^3)$), on choisit l'algorithme correspondant.

La consommation en mémoire peut être utilisée comme critère de choix supplémentaire, soit dans les cas où les algorithmes semblent tous deux être aussi rapides, soit dans les cas où il faut privilégier une faible consommation en mémoire. Ainsi, si les complexités $f_1(n)$ et $f_2(n)$ sont équivalentes, on calculera les consommations $m_1(n)$ et $m_2(n)$ en mémoire des deux algorithmes, et si une des consommations est strictement inférieure à l'autre, on choisira l'algorithme correspondant.

Sinon, on devra raffiner l'analyse de $m_1(n)$ et $m_2(n)$ ou utiliser d'autres critères (par exemple la facilité d'implémentation ou de maintenance). En pratique, on verra que les choses ne sont pas toujours aussi “mécaniques” et dépendent de ce qu'on veut privilégier, mais cette stratégie fournit un bon point de départ.

6.2 Mesure de la consommation en mémoire en Python

Le module `time` nous a fourni les outils nécessaires pour chronométrer des morceaux de code et mesurer en pratique le temps d'exécution de nos algorithmes. Pour mesurer en pratique la consommation en mémoire d'un algorithme, on utilisera la fonction `getsizeof` du module `sys` qui donne le nombre d'octets consommés par son paramètre.

Exemple 27. Voici quelques exemples d'utilisation de la fonction `getsizeof` :

```
>>> from sys import getsizeof
>>> getsizeof(10)
28
>>> getsizeof(10**10)
32
>>> getsizeof('a')
50
>>> getsizeof('bonjour')
56
```

On remarque qu'en Python, la consommation en mémoire d'un entier dépend de sa taille.

Quelques précautions sont de mise lors de l'utilisation de cette fonction : `getsizeof()` fonctionne comme on s'y attend pour les types basiques; mais pour les itérables comme les listes, il faut additionner les tailles des éléments et de la structure les contenant.

Exemple 28 (espace consommé par une liste)**.** Si l'on utilise `getsizeof` sur une liste, la taille renvoyée n'est pas celle à laquelle on s'attend :

```
>>> from sys import getsizeof
>>> a = 'structures de données et algorithmes'
>>> getsizeof(a)
109
>>> getsizeof([a])  # une liste contenant a coûte moins que a !
80
>>> getsizeof([1])
80
```

Pour en comprendre la raison, il est nécessaire d'examiner plus en détails comment les listes Python sont représentées : une liste n'est pas simplement une collection de variables en mémoire, mais toute une structure contenant entre autres :

— les éléments de la liste;

— un champ indiquant sa longueur (que `len` consulte pour la renvoyer en $O(1)$);

— et une "table des matières" indiquant pour chaque position de la liste où se trouve l'élément correspondant en mémoire.

L'appel à la fonction `getsizeof` sur une liste `L` ne renvoie pas la taille de chacun des éléments de la liste, mais seulement la taille de cette table des matières, ce qui explique par exemple pourquoi `getsizeof([10]) == getsizeof([10 ** 10])` même si les deux nombres de départ n'ont pas la même taille. Il faut donc ajouter à cette quantité la somme des tailles des éléments de la liste ... et bien entendu, si la liste contient d'autres itérables (comme dans le cas des matrices qui sont des listes de listes), on se retrouvera confronté au même problème pour ces éléments.

Nous allons donc écrire nos propres fonctions pour ces cas particuliers ; le résultat est montré à l'algorithme 15.

```python
from sys import getsizeof

def espace_liste(liste):
    """Renvoie la taille en octets d'une liste."""
    taille = getsizeof(liste)
    for element in liste:
        taille += getsizeof(element)
    return taille

def espace_matrice(matrice):
    """Renvoie la taille en octets d'une matrice."""
    taille = getsizeof(matrice)
    for ligne in matrice:
        taille += espace_liste(ligne)
    return taille
```

ALGORITHME 15: Calcul de l'espace mémoire consommé par une liste et par une matrice.

Remarquons que les nombres exacts renvoyés par `getsizeof` peuvent varier en fonction de la machine utilisée, du système d'exploitation installé, et même de la version de Python que l'on utilise. Ceci est un argument supplémentaire en faveur de l'utilisation de la notation $O(\cdot)$, puisque l'on peut difficilement prévoir la valeur exacte des constantes qu'on omet.

6.3 Moins de calculs, plus de mémoire

Examinons maintenant quelques techniques qui pourront nous permettre de gagner en rapidité en sacrifiant de la mémoire.

6.3.1 L'énumération

Il arrive régulièrement que l'on doive stocker des éléments dans une séquence, qu'elle soit implémentée sous la forme d'une liste ou d'un tableau. Ces structures sont très pratiques, mais présentent l'inconvénient que certaines opérations que l'on doit fréquemment réaliser sur ces structures prennent un temps proportionnel à leur nombre d'éléments. En particulier, le stockage des valeurs dans une structure S nécessite un espace en $O(|S|)$, mais :

— vérifier que $x \in S$ coûte $O(|S|)$ en temps ;

— supprimer x de S coûte entre $O(1)$ et $O(|S|)$ en temps (selon sa place dans la structure et le type de la structure utilisée) ;

— insérer x dans S coûte entre $O(1)$ et $O(|S|)$ en temps (selon sa place dans la structure et le type de la structure utilisée).

On peut améliorer le temps de calcul en procédant par **énumération** : l'idée est d'attribuer à chaque candidat x potentiel de S un champ indiquant s'il est présent dans la structure, et éventuellement en quelle quantité.

Exemple 29. Soit `L = [18, 5, 2, 9, 2, 7, 18, 17, 13, 18, 20, 17, 4, 12, 4]` une liste de n naturels entre 0 et $V = 20$. Au lieu de manipuler directement la liste `L`, on va créer une liste `C` de compteurs indiquant, pour chaque valeur `i` entre 0 et V compris, combien de fois cette valeur apparaît dans `L`. Pour l'exemple choisi, on obtient :

```
C = [0, 0, 2, 0, 2, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 2, 3, 0, 1]
```

Grâce à cette représentation, on peut simuler des opérations sur `L` à l'aide d'opérations sur `C`. Par exemple :

— l'opération `x in L` équivaut au test `C[x] != 0` ;

— l'opération `L.count(x)` équivaut à consulter `C[x]` ;

— l'opération `L.remove(x)` équivaut à l'opération `C[x] -= 1` si `C[x] > 0` ;

— ...

L'algorithme 16 montre une fonction renvoyant l'énumération d'un itérable de n naturels entre 0 et V, et s'exécute en $O(n+V)$. Le résultat est une liste de $V+1$ éléments contenant pour chaque position i le nombre d'occurrences de i dans l'itérable de départ.

```python
def enumerer(iterable_de_naturels):
    """Renvoie une liste dont la position i contient le nombre d'occurrences de i
    dans iterable_de_naturels. La taille de cette liste est celle du plus grand
    élément de l'ensemble plus 1."""
    if not iterable_de_naturels:  # max() plante sur un itérable vide
        return []

    compteurs = [0] * (max(iterable_de_naturels) + 1)

    for element in iterable_de_naturels:
        compteurs[element] += 1

    return compteurs
```

ALGORITHME 16: Énumération d'un itérable de naturels.

Avantages

Les opérations que l'on cherche à effectuer sur notre ensemble d'éléments sont plus rapides :

— pour savoir si $x \in S$, il suffit de consulter la case `compteurs[x]`, ce qui coûte un temps $O(1)$;

— pour ajouter x à S, il suffit d'incrémenter la case `compteurs[x]`, ce qui coûte un temps $O(1)$;

— pour retirer x de S, il suffit de décrémenter la case `compteurs[x]` ou de la mettre à 0, ce qui coûte un temps $O(1)$.

— enfin, il est possible de récupérer tous les éléments de l'ensemble dans l'ordre croissant *sans devoir le trier* : un simple parcours de liste par position croissante suffit.

Attention, les opérations ci-dessus supposent bien sûr que x est un naturel qui ne dépasse pas le maximum de S.

Exercice 12. Écrivez les fonctions d'insertion et de retrait d'un élément x dans la structure d'énumération de manière à ce que les cas où x dépasse le maximum de S fonctionnent aussi.

Inconvénients

En contrepartie de ces accélérations :

— l'espace consommé est proportionnel à la taille du plus grand élément de l'ensemble, puisqu'il faut prévoir un champ pour chaque candidat potentiel : si V est le maximum de S, alors l'espace supplémentaire utilisé est $O(V)$. Si l'on n'a pas prévu assez de place, on peut bien entendu toujours rallonger la liste en cours de route, mais il faut s'assurer que la mémoire disponible est suffisante.

— on ne peut pas représenter autre chose que des ensembles de naturels.

Exercice 13. Écrivez une fonction renvoyant l'énumération de S quel que soit le type des éléments de S.

Cette technique n'a bien sûr d'intérêt que si elle accélère les opérations qui nous intéressent. De plus, si l'ordre des éléments de la structure est important, on ne pourra pas l'appliquer non plus car la liste de compteurs dont on dispose ne nous donne pas d'informations sur les positions originales des éléments.

Application : le tri par énumération

L'énumération nous permet d'obtenir un algorithme de tri rapide dans le cas où les valeurs (naturelles) à trier appartiennent à un intervalle relativement restreint : au lieu d'avoir recours à l'un des algorithmes de tri en $O(n^2)$ de la section 3.3, ou à l'algorithme de tri en $O(n \log n)$ implémenté en Python, on peut facilement obtenir un algorithme de tri en $O(n+V)$, où V est la plus grande valeur apparaissant dans la liste.

L'idée est simple : il nous suffit d'énumérer la liste, ce qui nous fournit ses valeurs dans l'ordre croissant lorsqu'on parcourt la liste de compteurs résultante. Il n'y a donc plus qu'à recopier chaque élément le bon nombre de fois dans la liste de départ, en écrasant les données qu'elle contient. L'algorithme 17 montre une implémentation du tri par énumération basée sur l'algorithme 16.

Bien entendu, plus l'intervalle des valeurs possibles est grand, plus la complexité sera élevée. Remarquons que l'utilisation d'un dictionnaire de compteurs plutôt que d'une liste permet de limiter la complexité ; mais les choses se compliquent quand même, car stocker les compteurs sous la forme d'une liste nous permettait de récupérer les éléments dans le bon ordre, ce qui n'est plus le cas avec un dictionnaire.

```python
def tri_enumeration(T):
    """Trie la liste T par énumération."""
    decalage = 0

    for valeur, occurrences in enumerate(enumerer(T)):
        for i in range(occurrences):
            T[decalage + i] = valeur

        decalage += occurrences
```

ALGORITHME 17: Tri par énumération.

6.3.2 L'annotation et les listes "à trous"

L'**annotation** consiste à rajouter des informations aux données présentes dans notre structure. Une application directe de cette technique dans l'optique qui nous intéresse est la création d'une structure similaire aux listes mais dans laquelle on s'autorise des "trous", c'est-à-dire des positions qui ne contiennent pas de données réelles. On manipulera donc une structure contenant une liste d'éléments **et** une liste de positions libres.

Cette technique est intéressante dans les cas où l'on est obligé de conserver l'ordre des éléments (comme on l'a vu, l'énumération ne permet pas de conserver cet ordre), et où la structure à manipuler va être fréquemment modifiée par des insertions et des suppressions. En effet, la représentation obtenue nous permettra d'éviter les décalages causés par ces opérations en indiquant simplement que la case concernée ne contient plus de données (pour les suppressions) ou en n'utilisant que les cases libres sans décaler le reste (pour les insertions).

Exemple 30. Voici une liste de 16 éléments, avec 7 données "réelles" :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`donnees` :	12	3	31	54	7.5	19	4	47	5	-17	3	1	2	-15	-39	7
`libres` :	0	1	1	0	1	0	1	1	0	0	0	0	1	0	0	1

Certaines opérations que l'on voudrait effectuer directement sur `donnees` peuvent se traduire par des opérations sur `libres`. Par exemple :

— `donnees[i]` contient des données réelles si et seulement si `libres[i] != 0`;

— afficher les données réelles consiste à afficher les cases en position `i` pour lesquelles `libres[i] != 0`;

— pour supprimer `donnees[i]`, il nous suffit de faire `libres[i] = 1`;

— ...

Cette représentation est surtout rentable quand les données à manipuler sont grandes et que l'espace consommé est fixe. On veut pouvoir effectuer les mêmes opérations que sur les séquences déjà rencontrées, à savoir supprimer un élément, et insérer un élément sans perdre de données. Pour rappel, dans les listes, l'insertion et la suppression entraînaient des décalages, et ces décalages impliquaient que ces deux opérations étaient en $O(n)$. Ici, on peut s'en tirer de manière plus efficace en manipulant la liste `libres` : pour supprimer l'élément en position `i` dans les données, il nous suffit de mettre `libres[i]` à 1, ce qui se fait en temps constant car aucun décalage n'est requis. Pour insérer un élément `e` en position `i` dans les données :

1. si `libres[i] == 1`, il suffit de faire deux affectations : `donnees[i] = e` pour stocker `e`, et `libres[i] = 0` pour signaler que la position `i` n'est plus libre ;
2. si `libres[i] == 0`, on peut au choix :
   - refuser l'insertion et prévenir l'utilisateur,
   - ou chercher la position la plus proche de `i`, à gauche ou à droite, qui nous permette d'effectuer le moins de décalages nécessaires à l'insertion de e,
   - ou encore, si l'ordre d'insertion n'est pas important, se contenter de chercher une position libre utilisable.

La conversion d'une liste vers une liste à trous est simple : il suffit d'écrire une fonction recopiant une liste de n éléments donnée en entrée et renvoyant, en plus de la copie, une liste de n positions libres (toutes initialisées à 0 si l'on considère qu'aucune position n'est libre). Pour la conversion d'une liste à trous vers une liste classique, il nous suffit de recopier les éléments dont la position associée n'est **pas** libre et de renvoyer la liste résultante.

Le tableau ci-dessous compare les performances des opérations qui nous intéressent sur les deux structures :

Liste	**Classique**	**À "trous"**
insertion	$O(n)$	$O(n)$
suppression	$O(n)$	$O(1)$

Le prix des performances améliorées est l'espace supplémentaire requis, qui s'élève à $O(n)$ dans une implémentation naïve. En pratique, on peut réduire cette consommation à $O(\log n)$ avec une représentation binaire de `libres`.

Un exemple important d'utilisation de cette technique est la gestion des systèmes de fichiers sur un disque dur, où le coût des écritures (que ce soit pour enregistrer ou effacer des données) se fait particulièrement ressentir en pratique. Comme le disque dur ne changera jamais de capacité, on peut sans problème représenter chacun des secteurs dans l'annotation par une case dans une liste de données, et s'économiser des écritures inutiles en cas d'effaçage en stockant les positions des endroits libres (on accélère également ces opérations à l'aide de structures efficaces indexant les positions libres, pour ne pas devoir parcourir le disque en entier à chaque insertion). En contrepartie, cela signifie aussi qu'il restera sur le disque de nombreuses traces des données qu'on croyait avoir effacées...

6.3.3 Le partitionnement

De manière générale, le **partitionnement** consiste à découper les données reçues en entrée en classes d'équivalences de façon à les traiter plus efficacement. Le critère utilisé pour les équivalences varie selon l'application. Illustrons cette technique sur l'exemple simple suivant.

Exemple 31. Supposons que l'on doive écrire un programme qui devra fréquemment accéder aux mots d'un certain fichier texte `F`. Pour répondre à ce genre de requêtes de manière efficace, un prétraitement s'impose de toute façon puisqu'il est beaucoup plus rapide de parcourir une structure stockée en mémoire qu'un fichier sur le disque. Reste à savoir quelle structure de données utiliser pour obtenir de bonnes performances : on pourrait songer à construire l'ensemble de tous les mots du fichier, et à demander ensuite à Python si le mot renseigné appartient à l'ensemble construit, ce qui implique au pire cas de parcourir tous les mots existants.

Une manière plus intelligente de procéder consisterait à grouper les mots par taille, car il ne sert à rien de chercher un mot parmi un ensemble de mots n'ayant pas la même taille. C'est d'ailleurs la première vérification à laquelle on procède quand on doit vérifier si deux chaînes données sont les mêmes (voir l'algorithme 9).

On va donc partitionner l'ensemble des mots en ensembles de mots de même taille, que l'on structurera sous la forme d'un dictionnaire dont les clés sont les longueurs de mots possibles et dont chaque valeur correspond aux mots du fichier qui sont de cette longueur. Ainsi, au lieu d'écrire quelque chose comme "`mot in contenu_fichier`", on écrira plutôt quelque chose comme "`mot in annuaire[len(mot)]`". Remarquons que cela ne nous empêche pas d'en plus trier les mots d'une longueur donnée, afin de pouvoir tirer parti de la recherche dichotomique dans chaque sous-annuaire.

6.4 Moins de mémoire, plus de calculs

6.4.1 Compression de matrices symétriques

Certaines matrices dites *symétriques* satisfont la propriété suivante :

Définition 9. Une matrice $n \times n$ est **symétrique** si $\forall\ i, j \in \{1, 2, \ldots, n\}$: $\quad M[i][j] = M[j][i]$.

Exemple 32 (matrice symétrique)**.** Voici un exemple de matrice symétrique :

$$M = \begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 3 & 1 & 6 & 5 & 2 \\ 1 & 1 & 7 & 1 & 5 & 3 \\ 2 & 6 & 1 & 8 & 9 & 6 \\ 3 & 5 & 5 & 9 & 5 & 1 \\ 4 & 2 & 3 & 6 & 1 & 3 \end{array}$$

Intuitivement, on voit bien qu'on devrait pouvoir consommer moins d'espace que n^2 cases, car on stocke deux fois la même information au-dessus et en-dessous de la diagonale principale. L'idée va consister à représenter une matrice symétrique à l'aide d'une liste, en éliminant les éléments qui sont dupliqués.

En guise d'échauffement, commençons par examiner comment représenter n'importe quelle matrice par une simple liste [1].

Exemple 33. Voici une représentation d'une matrice M d'entiers sous la forme d'une liste T d'entiers :

$$M = \begin{array}{c|ccccc|} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 6 & 7 & 8 & 9 & 10 \\ 2 & 11 & 12 & 13 & 14 & 15 \\ 3 & 16 & 17 & 18 & 19 & 20 \\ 4 & 21 & 22 & 23 & 24 & 25 \\ \hline \end{array}$$

$$T = \begin{array}{|ccccccccccccccccccccccccc|} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 \\ \hline \end{array}$$

Le prix à payer est que l'on ne peut plus utiliser les opérateurs `[]` tels quels : l'élément `M[i][j]` se trouve en `T[i * n + j]` (ici, `n` est le nombre de colonnes), et on devra donc écrire `T[i * n + j]` partout où l'on a envie d'écrire `M[i][j]`.

Voyons maintenant comment l'on peut tirer parti de cette représentation particulière dans le cas des matrices symétriques : on va recopier les éléments de la matrice M qui se trouvent sous la diagonale, et oublier les autres.

Exemple 34. Si l'on ne garde que les éléments sous la diagonale dans une matrice symétrique, on obtient une liste plus compacte :

$$M = \begin{array}{c|ccccc|} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 3 & 1 & 6 & 5 & 2 \\ 1 & 1 & 7 & 1 & 5 & 3 \\ 2 & 6 & 1 & 8 & 9 & 6 \\ 3 & 5 & 5 & 9 & 5 & 1 \\ 4 & 2 & 3 & 6 & 1 & 3 \\ \hline \end{array}$$

$$T = \begin{array}{|ccccccccccccccc|} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \hline 3 & 1 & 7 & 6 & 1 & 8 & 5 & 5 & 9 & 5 & 2 & 3 & 6 & 1 & 3 \\ \hline \end{array}$$

Quel est le gain réalisé ? La matrice de départ contenait $n \times n = n^2$ éléments. La matrice compressée contient :

- le 1er élément de la première ligne,
- les 2 premiers éléments de la deuxième ligne,
- les 3 premiers éléments de la troisième ligne,

  $\vdots$
- les $n-1$ premiers éléments de la ligne $n-2$, et
- les n éléments de la ligne $n-1$.

1. Certains langages comme l'assembleur ne nous laissent pas le choix : on n'y trouve pas de listes ou de tableaux à plusieurs dimensions, et on est donc obligé d'avoir recours à cette représentation.

Autrement dit, le nombre d'éléments de T est

$$1+2+\cdots+n=\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

et on a donc réussi à gagner $n(n-1)/2$ cases.

En revanche, on ne peut plus utiliser les opérateurs `[]` tels quels pour deux raisons : on a une liste au lieu d'une matrice, et il "manque" des éléments par rapport à la matrice de départ. Comment accéder à l'élément `M[i][j]` dans `T` ?

On peut s'en tirer avec la même expression qui nous a servi à calculer le nombre d'éléments de T, et en se rappelant que l'on n'a stocké que les éléments sous la diagonale — autrement dit : tous ceux dont l'indice de colonne est inférieur ou égal à l'indice de ligne. Ainsi, lorsqu'on débute la lecture de la ligne `i` dans `M`, on a déjà stocké `i * (i+1) // 2` éléments dans `T`. Pour simuler le déplacement à la colonne j dans T, on doit donc accéder à `T[i * (i+1) // 2]`, qui correspond bien à `M[i][j]`. Comme dit plus haut, tout se passe bien si l'on reçoit deux indices valides i et j avec $j \leq i$; si jamais $j > i$, il nous suffit d'échanger les valeurs de i et j : par la définition 9, on tombera sur la même valeur.

L'écriture de la fonction d'accès aux éléments, qui renvoie la valeur équivalant à `M[i][j]`, s'écrit donc comme suit :

```
def acces_sym(T, i, j):
    """Renvoie la valeur M[i][j] d'une matrice symétrique
    M compressée dans une liste T."""
    if i < j:
        i, j = j, i
    return T[i * (i + 1) // 2 + j]
```

Pour l'affectation, on obtiendra la fonction suivante :

```
def affectation_sym(T, i, j, v):
    """Affecte la valeur M[i][j] d'une matrice symétrique
    M compressée dans une liste T."""
    if i < j:
        i, j = j, i
    T[i * (i + 1) // 2 + j] = v
```

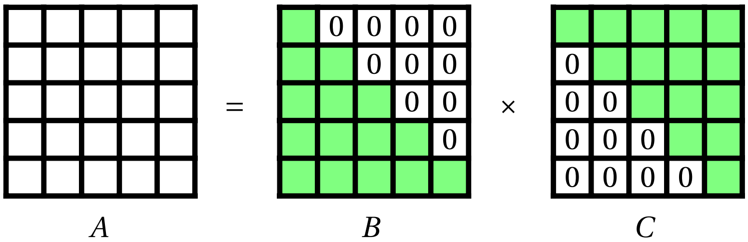
L'avantage principal de cette approche est l'objectif recherché : on a éliminé à peu près la moitié des éléments, ce qui n'est pas négligeable dans les applications où l'on a des matrices de grande taille. Les inconvénients sont la réécriture des opérations d'accès et d'affectation, qui sont plus compliquées mais se font encore en $O(1)$; et le fait que cette représentation est moins flexible que les matrices traditionnelles, puisqu'elle ne fonctionne que pour les matrices "toujours" symétriques (du début à la fin de l'exécution du programme).

Cette technique est surtout intéressante quand elle est implémentée de manière **transparente** – c'est-à-dire quand l'utilisateur ne s'en rend pas compte. Dans certains langages dont Python, on peut définir une classe "matrice" avec ses fonctions (ou opérateurs) d'accès et d'affectation [2]. Ce procédé permet d'avoir recours à la compression de matrice sans compli-

2. Les notions de "classe", "méthode", "objet" ... appartiennent à la programmation orientée objet et seront couvertes dans d'autres cours.

quer la vie de l'utilisateur.

Parmi les exemples de situations où l'on pourrait utiliser cette technique, on retrouve la **factorisation LU** (voir par exemple Schwarzenberg-Czerny [1] pour plus de détails), qui consiste à représenter une matrice carrée par le produit de deux matrices "triangulaires" :



Le calcul de cette factorisation facilite la résolution de systèmes d'équations linéaires et accélère également certains calculs (inverse de A, etc.). Il n'est pas nécessaire de stocker autre chose que les éléments non-nuls, et l'on peut donc appliquer la méthode utilisée pour compresser les matrices symétriques pour éviter de stocker les éléments nuls.

6.5 Remarques

On a parlé dans le chapitre 2 de la classe de complexité P, qui regroupe l'ensemble des problèmes que l'on peut résoudre en temps polynomial en la taille des données en entrée. Une classe de complexité analogue existe du point de vue de la complexité spatiale : la classe PSPACE, qui contient tous les problèmes que l'on peut résoudre en consommant un espace en mémoire polynomial en la taille des données en entrée. Remarquons que cela ne nous dit rien sur le *temps* mis pour les résoudre, tout comme la classe P ne nous dit rien sur la consommation éventuelle en mémoire. Ce qui nous mène à un autre problème ouvert en informatique théorique :

Est-ce que P = PSPACE **?**

Il n'est pas difficile de voir que l'inclusion P ⊆ PSPACE est vérifiée : si un problème soluble en temps polynomial nécessitait un espace mémoire "superpolynomial" (exponentiel par exemple), alors il nous faudrait au moins comptabiliser le temps nécessaire pour réserver cet espace en mémoire, qui ne serait pas non plus polynomial et le problème mentionné ne pourrait donc pas être dans P.

Bibliographie

[1] A. SCHWARZENBERG-CZERNY, *On matrix factorization and efficient least squares solution*, Astronomy and Astrophysics Supplement, 110 (1995), pages 405–410.

Chapitre 7

Récursivité

Sommaire

La **récursivité** est la propriété que possède un objet ou un concept de s'exprimer "en fonction de lui-même". En algorithmique et en programmation, une fonction dont l'appel peut conduire à sa propre invocation est qualifiée de **récursive**.

Exemple 35. Voici une fonction récursive (mauvaise ; on expliquera pourquoi plus loin) :

```python
def f(x):
    a = f(2 * x - 17)
    return (5 * a + 18)
```

Le compilateur (ou l'interpréteur) doit donc "comprendre" que l'appel à `f` fait partie de la définition de cette fonction.

La récursivité est un concept important en algorithmique : pour écrire des algorithmes, mais également pour décrire ou définir des structures de données. Cette technique rend le raisonnement et l'écriture de code plus facile. Elle a son lot d'avantages, mais également d'inconvénients dont on parlera un peu plus loin.

Parmi les exemples d'algorithmes s'exprimant simplement, on retrouve celui de la recherche de fichiers, présenté ici de manière informelle :

1. si le fichier à chercher est dans le répertoire actuel, on renvoie son chemin ;
2. sinon, on le cherche dans les sous-répertoires.

7.1 Principes de base

Définition 10. Une fonction est **récursive** si son exécution *peut* conduire à sa propre invocation.

Une telle fonction se présente donc comme suit :

```
def f(P):       # P = liste de paramètres
    # instructions (1)
    x = f(Q)    # appel avec d'autres paramètres
    # instructions (2)
    return resultat
```

Le renvoi de valeur n'est pas obligatoire et dépendra bien sûr du problème à résoudre ; on verra des exemples de fonctions récursives qui ne renvoient rien.

Comme avec les boucles, il est facile d'écrire des fonctions qui ne s'arrêtent jamais ; et pour rappel : un algorithme *correct* se termine ! Il faut donc toujours s'assurer que la fonction récursive qu'on écrit arrête à un moment de s'appeler. Une fonction récursive *correcte* doit donc posséder une **condition d'arrêt**, qui détermine dans quelle(s) situation(s) les appels récursifs doivent cesser :

```
def f(P):
    if test(P):  # condition d'arrêt
        # bloc sans appel récursif
        return resultat_1

    # bloc avec appel(s) récursif(s)
    return resultat_2
```

... et il faut bien sûr s'assurer qu'elle finira par être vraie ! Pour que la condition d'arrêt soit vraie, il faut que les appels récursifs utilisent d'autres paramètres que l'appel initial.

Exemple 36. Voici deux fonctions récursives visant à réaliser la même tâche :

Mauvaise fonction récursive

```
def f(n): # n naturel
    if n == 0:
        print("!")
    else:
        print("*", end="")
        f(n)
```

Bonne fonction récursive

```
def f(n): # n naturel
    if n == 0:
        print("!")
    else:
        print("*", end="")
        f(n-1)
```

La première fonction ne s'arrête jamais car si n ne vaut pas 0 au début, il ne vaudra jamais 0.

- *(fin exemple 36)* - -

En résumé, le canevas d'une fonction récursive correcte se ramène au suivant :

```
def f(P): # P = paramètre(s)
    if test(P):  # condition d'arrêt
        # bloc sans appel récursif
    else:
        # bloc avec appel(s) récursif(s) sur des paramètres "plus simples"
```

La signification de "plus simple" varie selon le contexte :

— si P est un naturel et que la condition d'arrêt se base sur des petits nombres, alors $P' < P$;

— si P est une liste et que la condition d'arrêt vérifie si la liste est vide, alors P' est une liste plus petite ;

— ...

7.2 Algorithmes récursifs basiques

Dans le cas le plus simple, le problème à résoudre est déjà formulé de manière récursive, et il suffit alors de traduire cette formulation en un algorithme en pseudocode (ou dans le langage de programmation choisi). Si ce n'est pas le cas :

1. on cherche d'abord une formulation récursive ;
2. on s'assure que cette formulation possède une ou plusieurs conditions d'arrêt ;
3. et après seulement, on écrit l'algorithme correspondant.

Illustrons cette méthode sur quelques exemples.

7.2.1 Nombres de Fibonacci

Un exemple très simple de cas où l'on peut directement obtenir un algorithme récursif est celui des nombres de Fibonacci. Pour rappel, le $n^{\text{ème}}$ nombre de Fibonacci F_n est donné par :

$$\forall\, n \in \mathbb{N} : \quad F_n = \begin{cases} n & \text{si } n \leq 1, \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

La définition de ces nombres est récursive, et l'on nous fournit également la condition d'arrêt. Il ne nous reste plus qu'à traduire cette définition en un algorithme, et l'on obtient directement l'algorithme 18.

7.2.2 Factorielle récursive

Rappelons que la **factorielle** d'un nombre naturel n est

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1,$$

```
def fiboR(n):
    if n < 2:
        return n
    return fiboR(n - 1) + fiboR(n - 2)
```

ALGORITHME 18: Calcul récursif des nombres de Fibonacci.

avec le cas particulier $0! = 1$. On sait déjà comment la calculer de manière itérative (voir page 11), et l'on va maintenant tenter de la calculer de manière récursive en appliquant la méthodologie donnée au début de la section 7.2.

1. **trouver une formulation récursive :** en écrivant la définition, on se rend compte que :

$$n! = n \times \overbrace{(n-1) \times (n-2) \times \cdots \times 2 \times 1}^{=(n-1)!}.$$

   On en déduit donc que $n! = n \times (n-1)!$.

2. **trouver une condition d'arrêt :** elle nous est directement fournie dans la définition originale : si $n = 0$, alors la factorielle correspondante vaut 1. Pour calculer tous les autres cas, on peut avoir recours à la formule de récurrence que l'on vient de trouver. On peut donc réécrire la définition de manière récursive :

$$\forall\, n \in \mathbb{N} : \quad n! = \begin{cases} 1 & \text{si } n \leq 1, \\ n \times (n-1)! & \text{sinon.} \end{cases}$$

3. **traduire la définition en code :** cette étape de traduction est la plus simple si la définition obtenue est suffisamment claire ; c'est le cas ici, et l'on en déduit aisément l'algorithme 19.

```
def factoR(n):
    """Renvoie la factorielle du naturel n."""
    if n <= 1:
        return 1
    return n * factoR(n - 1)
```

ALGORITHME 19: Calcul récursif de la factorielle.

Le regroupement des cas $n = 0$ et $n = 1$ ne faisait pas partie de la définition initiale ; dans la définition récursive que l'on a construite, ce traitement unifié permet d'économiser un appel récursif pour le cas $n = 1$. Ceci n'aura aucun impact en pratique, mais comme on le verra plus loin, les appels récursifs peuvent s'avérer coûteux en temps d'exécution et en consommation mémoire : il est donc avantageux de pouvoir minimiser le nombre de ces appels lorsque c'est possible.

7.2.3 Puissance récursive

On a déjà vu comment calculer a^n de manière itérative dans le cas où $n \in \mathbb{N}$. Pour pouvoir calculer ce nombre de manière récursive, appliquons la méthodologie que l'on a suivie dans le cas de la factorielle :

1. **trouver une formulation récursive :** en écrivant la définition de la puissance, la définition récursive apparaît naturellement :

$$a^n = \overbrace{a \times a \times \cdots \times a}^{n \text{ termes}} = a \times \overbrace{a \times a \times \cdots \times a}^{n-1 \text{ termes}} = a \times a^{n-1}.$$

2. **trouver une condition d'arrêt :** comme n est un nombre naturel, la plus petite valeur qu'il peut prendre est 0. Dans ce cas, on sait que $a^0 = 1$ quelle que soit la valeur de a, et ceci peut se calculer sans appel récursif. Notre définition récursive est maintenant complète :

$$\forall\, a \in \mathbb{R}, \forall\, n \in \mathbb{N} : \quad a^n = \begin{cases} 1 & \text{si } n = 0, \\ a \times a^{n-1} & \text{sinon.} \end{cases}$$

3. **traduire la définition en code :** on déduit facilement le code à écrire de la définition que l'on vient d'obtenir : si $n = 0$, on renvoie 1, sinon on renvoie a multiplié par a^{n-1}, que l'on calcule avec un appel récursif (voir l'algorithme 20).

```python
def puissR(a, n):
    """Renvoie le réel a élevé à la puissance naturelle n."""
    if n == 0:
        return 1
    return a * puissR(a, n - 1)
```

ALGORITHME 20: Calcul récursif d'une puissance.

7.3 Coût en temps et en mémoire des algorithmes récursifs

On a vu comment calculer la complexité des algorithmes itératifs. Les règles ne changent pas dans le cas des algorithmes récursifs ; mais ici, le nombre "d'itérations" est remplacé par le nombre d'appels récursifs. Les variantes récursives de la factorielle ou de la puissance ne sont donc pas plus rapides d'un point de vue $O(\cdot)$: les algorithmes 19 et 20 sont donc en $O(n)$, puisqu'ils effectuent au pire $O(n)$ appels récursifs à une fonction dont tous les autres calculs sont en $O(1)$. Pour le calcul récursif des nombres de Fibonacci, la réponse semble moins évidente ; les *arbres d'appels* que nous allons voir ci-dessous nous aideront à la trouver.

Cependant, on a aussi des coûts "cachés" liés à l'usage de la récursivité. En effet, à chaque appel de fonction, on doit sauvegarder un *contexte* qui nous permet de revenir en arrière dans les appels récursifs, et on consomme donc de l'espace mémoire supplémentaire directement proportionnel au nombre d'appels réalisés. C'est pourquoi il importera d'être prudent quand on décide de réaliser des fonctions récursives sur des structures plus grandes et plus complexes que de simples entiers (listes, ensembles, ...).

7.3.1 Arbres d'appels

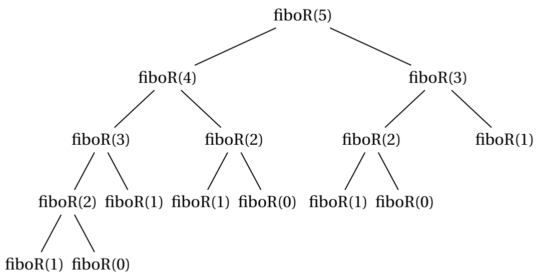
Les **arbres d'appels** permettent de visualiser ce qui se produit quand on appelle une fonction (récursive ou non). Ils aident aussi à évaluer la complexité de l'algorithme correspondant.

Définition 11. Soit $f(\cdot)$ une fonction récursive appelée avec les paramètres P. Soit P'_1, P'_2, ..., P'_k les paramètres des k appels récursifs réalisés lors de l'exécution de $f(P)$. L'**arbre d'appels** de $f(P)$ est un nœud étiqueté par $f(P)$ ayant pour descendants les arbres d'appels de $f(P'_1)$, $f(P'_2)$, ..., $f(P'_k)$. Un nœud n'ayant pas de descendant est qualifié de **feuille** et correspond à un appel où la condition d'arrêt est vérifiée.

L'énumération des nœuds de l'arbre d'appels nous permet d'évaluer la complexité des fonctions récursives. À titre d'exemple, examinons l'arbre d'appels pour le calcul récursif du cinquième nombre de Fibonacci.

Exemple 37. Si l'on exécute le code montré ci-dessous à gauche avec la valeur 5 pour n, on obtient l'arbre d'appels montré à droite. On y voit que le calcul de F_5 nécessite le calcul de F_4 et de F_3, que le calcul de F_4 nécessite celui de F_3 et de F_2, et ainsi de suite.

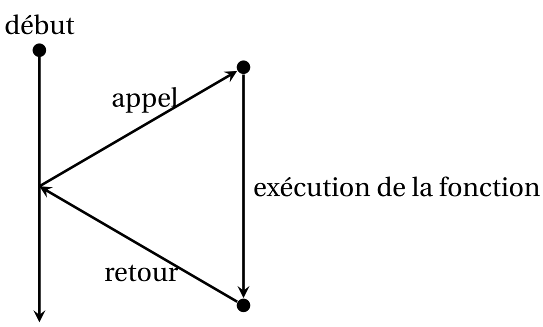
```
def fiboR(n):
    if n < 2:
        return n
    return fiboR(n - 1) + fiboR(n - 2)
```



Pour $n = 5$, on a donc 15 appels à effectuer. Comme chaque appel effectue deux appels récursifs, on se retrouve avec une complexité de ... $O(2^n)$! Dans ce cas précis, il est donc préférable d'utiliser la version itérative puisqu'elle est en $O(n)$.

7.3.2 Contexte et *stack frames*

Le **contexte** d'une fonction est l'ensemble des variables (et de leurs valeurs) qu'elle utilise. Lorsqu'on appelle une fonction, on interrompt le flux du programme :



Il faut donc sauvegarder les données et l'état quelque part de manière à pouvoir les récupérer lorsque la fonction se termine ; ces informations seront stockées sur la **pile**, un espace en mémoire destiné à cet usage et structuré comme tel. En particulier, chaque appel récursif nous obligera à sauvegarder le contexte, comme le montre la Figure 7.1. Ceci a également un impact sur la complexité spatiale de l'algorithme : si la fonction $f(\cdot)$ utilise une nouvelle variable entière dans sa définition, qui coûte un espace constant, mais effectue n appels récursifs, alors sa complexité spatiale sera en $O(n)$ [1].

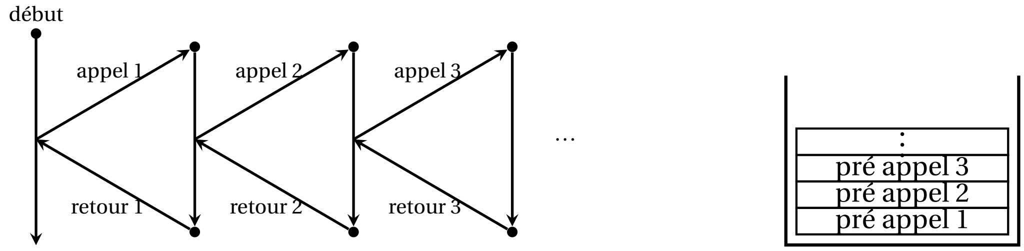


FIGURE 7.1 – Sauvegarde des contextes lors des appels récursifs.

7.3.3 Limitations pratiques

La pile mentionnée ci-dessus possède une taille limitée ; cela signifie donc que Python limite le nombre d'appels récursifs que l'on peut effectuer.

Exemple 38 (trop d'appels récursifs)**.** Si l'on exécute la fonction de l'algorithme 19, tout se passera bien jusqu'à une certaine valeur :

```
>>> factoR(997)
# ok, le résultat s'affiche
>>> factoR(998)
...
RuntimeError: maximum recursion depth exceeded in comparison
```

Le message ci-dessus signifie que l'on a effectué trop d'appels récursifs : soit parce que le code est correct mais qu'on l'a exécuté sur quelque chose de trop grand pour arriver au bout des appels, soit parce que le code est erroné. Si l'on voit ce message même pour des données de petite taille, cela veut généralement dire qu'on a oublié une condition d'arrêt ou qu'un des appels récursifs est incorrect.

7.4 Fonctions auxiliaires

Comme on le verra dans la suite avec les fonctions récursives sur des itérables, il est parfois utile de disposer d'une **fonction auxiliaire** dont le seul rôle sera d'appeler une fonction récursive avec les bons paramètres. Ce genre de besoin se manifeste quand il est nécessaire dans les appels récursifs de faire une différence entre l'appel initial de la fonction et ceux qui seront provoqués par la fonction elle-même, ou encore quand on a besoin de paramètres supplémentaires qu'il nous faut initialiser sans l'intervention de l'utilisateur. Dans ce cas, au lieu d'écrire une fonction récursive agissant directement sur les paramètres donnés, on écrira une fonction auxiliaire se présentant comme suit et qui appellera la fonction récursive souhaitée :

1. Ces estimations varient en fonction du type des variables et du langage utilisé.

```python
def fonction_auxiliaire(parametres_originaux):
    # initialisation de paramètres supplémentaires
    fonction_recursive(parametres_originaux, parametres_supplementaires)
```

Exemple 39. Supposons qu'on doive afficher récursivement les nombres de 1 à n, en donnant comme information supplémentaire à chaque ligne la quantité de nombres qu'il nous reste à afficher. Cette quantité dépend du nombre qu'on nous a donné au départ et de la profondeur actuelle dans les appels récursifs; la première information étant connue mais pas la seconde, il va nous falloir créer une fonction auxiliaire intermédiaire qui lancera la fonction récursive avec la bonne quantité.

Commençons par écrire la fonction récursive : si n vaut 0, on n'a rien à faire; sinon, on génère les $n-1$ nombres précédents récursivement, et l'on affiche ensuite le nombre n. De plus, il nous faut afficher la quantité de nombres restants. On suppose que cette quantité a été initialisée pour nous, et on peut donc l'afficher en même temps que n; dans les appels récursifs par contre, il nous faut augmenter cette quantité de 1 car elle représentera le nombre d'appels dont on doit revenir. On obtient donc ce qui suit :

```python
def afficher_nombres_rec(n, restants):
    if n:
        afficher_nombres_rec(n-1, restants+1)
        print(n, '(plus que', restants, 'nombres)')
```

Il ne nous manque plus qu'une fonction auxiliaire qui lancera l'appel initial et initialisera la quantité de nombres restants à la bonne valeur. Comme on aura terminé le travail en revenant à l'appel initial, il ne nous restera plus de nombres à afficher, et la valeur initiale de `restants` sera donc nulle :

```python
def afficher_nombres(n):
    afficher_nombres_rec(n, 0)
```

L'exécution du code pour $n = 5$ nous donne ceci :

```
>>> afficher_nombres(5)
1 (plus que 4 nombres)
2 (plus que 3 nombres)
3 (plus que 2 nombres)
4 (plus que 1 nombres)
5 (plus que 0 nombres)
```

7.5 Algorithmes récursifs sur des séquences

On a vu quelques exemples d'algorithmes récursifs sur des nombres naturels. Rien ne nous empêche d'appliquer la récursivité à des objets plus complexes, comme des listes ou des matrices. La stratégie présentée en section 7.2 ne change pas fondamentalement, mais il sera souvent utile de voir les objets manipulés également comme des structures récursives.

Prenons par exemple le cas des listes; on peut les voir de deux façons :

1. sous la forme de séquences : c'est le point de vue adopté par les algorithmes itératifs vus jusqu'ici, qui les parcourent position par position ;
2. sous une forme récursive : ceci est possible de deux manières :
   (a) une liste est soit une liste vide, soit une liste suivie d'un élément (Figure 7.2(a)) ; ou
   (b) une liste est soit une liste vide, soit un élément suivi d'une liste (Figure 7.2(b)).

L'interprétation récursive des listes sera essentielle pour aborder les problèmes à résoudre sur les listes de manière récursive. L'une ou l'autre des deux interprétations récursives (liste + élément ou élément + liste) sera, selon le problème examiné, plus simple à utiliser.

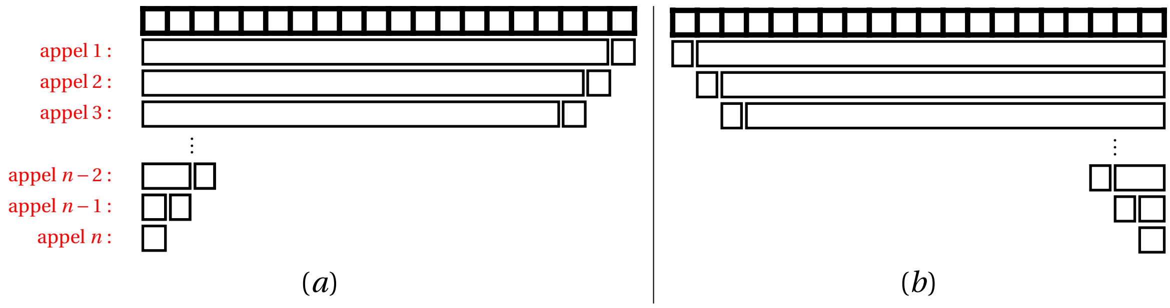


FIGURE 7.2 – Les découpes d'une liste qu'effectue une fonction récursive à chaque appel.

Pour illustrer ce principe, essayons d'implémenter la recherche linéaire vue dans le chapitre 3 de façon récursive. La stratégie de la section 7.2 s'applique également ici :

1. **trouver une formulation récursive :** notre objectif est de renvoyer la position d'un élément e dans une liste L, que l'on voit de manière récursive. Donc :
   (a) si l'on voit la liste L comme un élément x suivi d'une liste L', on renvoie la position de x si $x = e$; si $x \neq e$, on renvoie le résultat de la recherche de e dans L' ;
   (b) si l'on voit la liste L comme une liste L'' suivie d'un élément y, on renvoie la position de y si $y = e$; si $y \neq e$, on renvoie le résultat de la recherche de e dans L''.

   Le principe est donc identique dans les deux cas, mais l'implémentation de l'algorithme sera légèrement différente selon la vision adoptée.
2. **trouver une condition d'arrêt :** elle nous est directement fournie dans la vision récursive d'une liste : soit la liste est un élément suivi d'une liste (ou une liste suivie d'un élément), soit elle est vide. Si elle est vide, elle ne peut pas contenir l'élément e et il n'y a donc plus rien à faire.

La dernière étape, à savoir celle de la traduction de l'algorithme en code, nécessite plus de prudence dans le cas des listes. En effet, comme on va le voir, les choix irréfléchis vont directement influer négativement sur les performances.

7.5.1 Première tentative : utilisation de *slices*

L'approche la plus simple, mais également la plus mauvaise, consiste à utiliser des *slices* :

```python
def recherche_recursive_slices(liste, element):
    """Renvoie la position de l'élément dans la liste, ou None si elle ne le
```

```
    contient pas."""
    if liste:  # si L est vide, on abandonne
        if element == liste[-1]:
            return len(liste) - 1
        return recherche_recursive_slices(liste[:-1], element)
```

On a ici adopté la représentation de la Figure 7.2(a), qui nous permet dans les appels récursifs de ne pas devoir nous soucier d'adapter les indices. On voit qu'ici, la fonction s'appelle elle-même au lieu de parcourir explicitement la liste avec une boucle comme dans la version itérative.

Cette implémentation fonctionne mais est lente : chaque appel récursif crée un *slice*, et chaque *slice* est une copie de la liste de départ (de taille n, puis $n-1$, puis $n-2$, ...). Dès lors, la consommation en mémoire est en $O(n^2)$! De plus, la recherche elle-même sera en $O(n^2)$, puisque copier une (partie de) liste est une opération de complexité linéaire en la taille de la liste.

7.5.2 Seconde tentative : utilisation d'indices

Pour régler ces problèmes, une option plus intelligente consiste à utiliser un indice à incrémenter ou à décrémenter dans les appels récursifs de manière à progresser dans la liste. On obtiendrait donc le code suivant :

```
def recherche_recursive(liste, element, position):
    """Renvoie la position de l'élément dans la liste s'il se trouve entre le
    début et la position donnée, ou None dans le cas contraire."""
    if position < 0:  # condition d'arrêt
        return None
    if liste[position] == element:
        return position
    return recherche_recursive(liste, element, position - 1)
```

On y a réglé les deux problèmes de la version précédente : il n'est plus nécessaire d'utiliser des slices puisque l'indice `position` parcourt la liste dans les appels récursifs, et on n'a plus besoin de la fonction `len`. L'inconvénient restant est que l'utilisateur doit communiquer la valeur initiale de `position` à la fonction; pour éviter cet inconfort, on fournira en général une fonction auxiliaire que l'utilisateur appellera et qui se chargera de ce travail :

```
def recherche_element(liste, element):
    """Renvoie la position de l'élément dans la liste, ou None si elle ne le
    contient pas."""
    return recherche_recursive(liste, element, len(liste) - 1)
```

Enfin, remarquons que les *slices* sont une particularité du langage Python, et il n'est pas du tout garanti que l'on retrouve cette fonctionnalité dans un autre langage; en revanche, la deuxième approche que l'on vient de voir pourra se traduire sans problème dans un autre langage.

7.6 Cas plus complexes

Les problèmes que l'on a résolus jusqu'ici de manière récursive étaient des exemples simples qui nous ont plus servi d'entraînement que de cas où la récursivité était réellement essentielle. La récursivité montre toute sa puissance dans des situations où s'en passer mènerait à l'écriture d'un code très complexe ou difficilement généralisable (la recherche de fichiers sur un disque dur en était un bon exemple). À titre d'exemple, examinons quelques problèmes que l'on peut facilement résoudre récursivement, mais difficilement de manière itérative.

7.6.1 Générer tous les mots binaires

On sait qu'il existe 2^n mots binaires sur n bits ; mais comment peut-on les afficher tous, plutôt que simplement leur nombre ? Plusieurs options s'offrent à nous pour résoudre ce problème de manière itérative, mais une approche récursive du problème nous permet de le résoudre très simplement. Pour la trouver, appliquons la méthodologie présentée en section 7.2.

1. **trouver une formulation récursive** : on peut décomposer un mot binaire comme on a décomposé une liste dans la Figure 7.2(b) : un mot binaire sur n bits est un bit suivi d'un mot binaire sur $n-1$ bits.

   Il nous faut ensuite trouver une approche permettant de générer récursivement tous ces mots binaires ; pour ce faire, il nous suffit de générer tous les mots binaires sur n bits commençant par un 0, puis tous ceux commençant par un 1. Pour remplir le restant du mot, rappelons-nous qu'il s'agit d'un mot binaire sur $n-1$ bits, et qu'on peut donc le remplir de la même manière.
2. **trouver une condition d'arrêt** : on arrête la génération quand on a généré tous les mots, et on arrête de construire un mot quand on a rempli les n bits disponibles.
3. **traduire la formulation en code** : afin de réduire l'espace consommé en mémoire, on utilisera une seule liste de n éléments pour stocker le mot que l'on veut générer : comme on se contente d'afficher ces mots sans rien en faire d'autre ensuite, il est donc inutile de les conserver. Pour faciliter la tâche à l'utilisateur de notre fonction, qui n'a pas besoin de connaître le fonctionnement de notre algorithme, on écrira donc deux fonctions : la première initialise une liste de longueur n que l'on modifiera et dont on affichera le contenu ; la seconde, qui sera notre fonction récursive appelée par la première, génèrera et affichera les mots binaires. L'algorithme 21 montre le résultat.

Outre la génération de ces mots proprement dits, on pourrait utiliser cet algorithme pour générer tous les sous-ensembles d'un ensemble donné : en effet, un `0` en position `i` indique qu'on ne sélectionne pas l'élément correspondant, tandis qu'un `1` indiquerait qu'on le sélectionne. Il nous suffirait ensuite d'afficher les éléments dont la case correspondante est à `1` plutôt que le mot binaire généré.

Exercice 14. Adaptez l'algorithme 21 pour qu'il génère tous les mots sur n'importe quel alphabet fourni en paramètre.

```python
def generer_mots_binaires(n):
    '''Initialise une liste de n caractères et appelle la fonction de
    génération.'''
    liste = ['0'] * n
    generer_et_afficher_mots(liste, 0, n)


def generer_et_afficher_mots(liste, position, taille):
    '''Génère et affiche les mots binaires sur taille bits. La liste fournie est
    modifiée et contient le mot à afficher.'''
    # le dépassement de la taille conclut la génération du mot actuel; on
    # l'affiche et on s'arrête
    if position >= taille:
        print(''.join(liste))
        return

    # générer tous les mots contenant un 0, puis un 1, à la position donnée
    for valeur in ('0', '1'):
        liste[position] = valeur
        generer_et_afficher_mots(liste, position+1, taille)
```

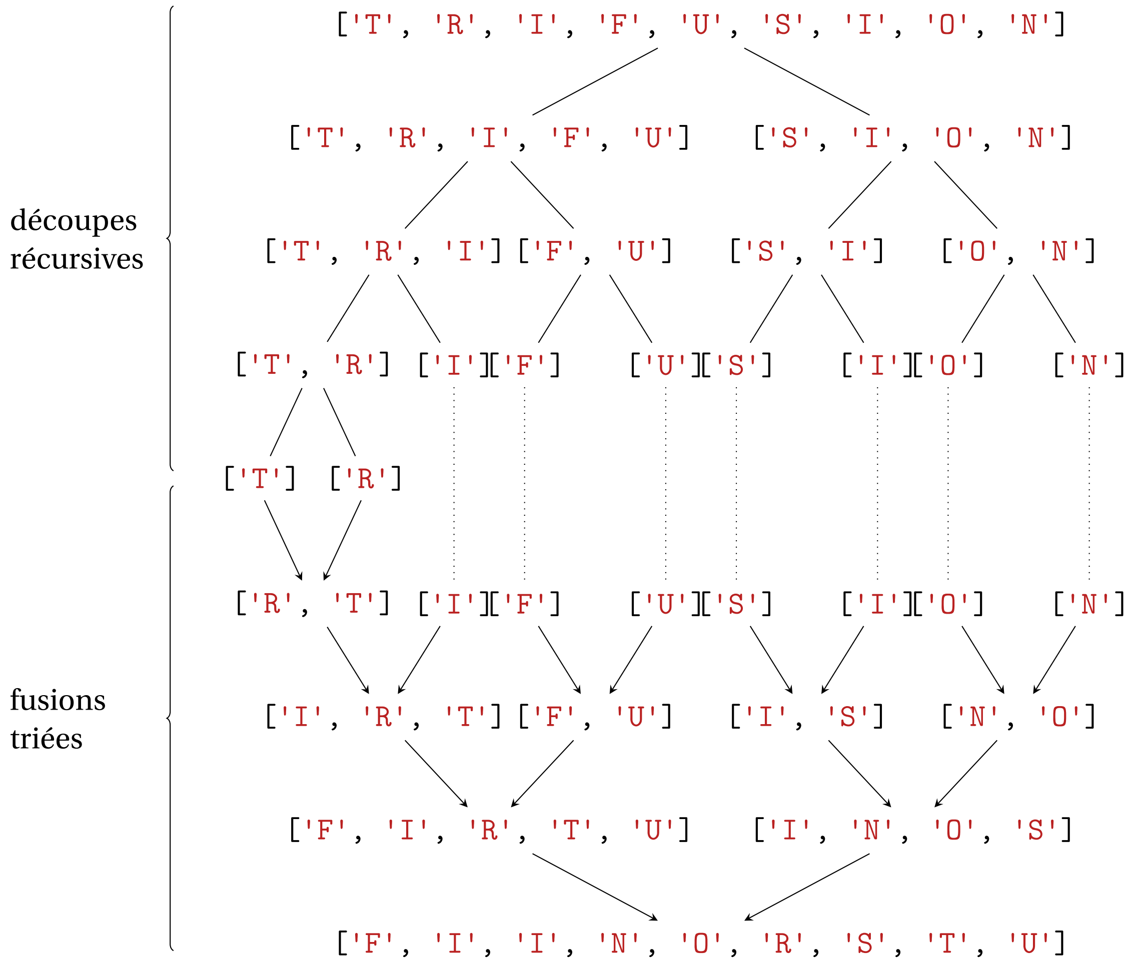
ALGORITHME 21: Génération récursive des mots binaires sur n bits.

7.6.2 Le tri fusion

La récursivité nous permet de décrire de manière relativement simple un nouvel algorithme de tri : le *tri (par) fusion*, qui illustre parfaitement une stratégie connue sous le nom de “diviser pour régner”.

L'idée du tri fusion est la suivante : pour trier une liste, il nous suffit de trier ses deux moitiés séparément, puis de les fusionner de façon à ce que le résultat soit trié. Les découpes se poursuivent récursivement jusqu'à ce qu'il ne soit plus possible de les réaliser (c'est-à-dire jusqu'à ce qu'on tombe sur une liste contenant au plus un élément).

Exemple 40 (adapté de Skiena [1])**.** Voici les étapes de l'algorithme du tri fusion appliqué à la liste `['T', 'R', 'I', 'F', 'U', 'S', 'I', 'O', 'N']` :



- *(fin exemple 40)* - -

Les découpes

Le tri se fait comme suit : on trie récursivement la première moitié de la liste, puis la seconde moitié de la liste, et l'on fusionne ensuite ces deux moitiés de manière triée. Il nous faut bien entendu une condition d'arrêt pour cette approche récursive, qui sera la suivante : si la taille de la liste à trier est inférieure à 2, il n'y a rien à faire.

Les sous-listes montrées dans l'exemple 40 sont là pour simplifier la compréhension de l'algorithme ; en réalité, on ne va pas recopier ces sous-listes puisqu'on a déjà vu que cela coûterait cher en mémoire (cf. sous-section 7.5.1). On va donc plutôt utiliser des indices de début et de fin pour délimiter les parties de la liste qui nous intéressent. Ceci nous donne donc l'algorithme ci-dessous, implémenté à l'aide d'une fonction auxiliaire :

Les fusions

Soit `lst` une liste dont on a isolé une portion à l'aide de trois indices : `debut`, `milieu` et `fin`. On suppose que `lst[debut:milieu]` et `lst[milieu:fin+1]` sont triées, et notre but est qu'à la fin de l'exécution de notre algorithme de fusion, `lst[debut:fin+1]` le soit aussi.

On veut en outre que l'étape de fusion soit rapide, c'est-à-dire qu'elle s'exécute en temps

```python
def tri_fusion(liste):
    """Fonction auxiliaire appelant le tri par fusion récursif."""
    tri_fusion_rec(liste, 0, len(liste)-1)

def tri_fusion_rec(lst, debut, fin):
    """Trie récursivement la liste lst[debut:fin+1] par fusion."""
    # plus que 0 ou 1 élément à trier
    if debut >= fin:
        return

    # partitionner lst en deux-sous listes et les trier
    milieu = (debut + fin) // 2
    tri_fusion_rec(lst, debut, milieu)
    tri_fusion_rec(lst, milieu+1, fin)

    # fusionner les deux sous-listes triées
    fusionner(lst, debut, milieu, fin)
```

ALGORITHME 22: Tri fusion.

linéaire en la taille de la liste donnée en entrée. Pour ce faire, on va utiliser une idée rappelant ce qu'on a couvert dans le cas du tri pair / impair en section 3.4 : deux curseurs qui se déplaceront simultanément dans `lst[debut:milieu+1]` et `lst[milieu:fin+1]`, et qui nous permettront à chaque étape de recopier le plus petit des deux éléments dans une liste auxiliaire notée aux.

Complexité du tri fusion

La représentation graphique de l'exemple 40 nous aide à raisonner pour le calcul de la complexité du tri par fusion : il faut se demander, à chaque "niveau" de l'arbre de découpe, quel est le temps de calcul que l'on consomme pour effectuer les fusions. Ensuite, la connaissance de la "profondeur" de l'arbre de découpe (son nombre de niveaux) nous permettra de conclure.

1. Comme nous avons réussi à écrire un algorithme de fusion s'exécutant en temps linéaire, le temps de calcul consommé au niveau 1 implique de fusionner deux listes triées de taille $n/2$ en une liste triée de taille n, ce que l'on effectue en temps $O(n)$. De même, au niveau 2, on a quatre listes de taille $n/4$ à fusionner deux à deux en deux listes de taille $n/2$, ce qui nécessitera également un total de n opérations, et ainsi de suite. Le travail de fusion réalisé à chaque niveau de l'arbre est donc de l'ordre de $O(n)$.
2. Pour deviner quelle sera la profondeur de l'arbre, il est utile, comme dans l'analyse de la dichotomie (cf. sous-sous-section 3.2.2), de supposer que la liste à trier est de taille $n = 2^k$ et de se demander combien de fois il faudra la couper en deux jusqu'à ce que ce ne soit plus possible. Par le même genre de raisonnement que pour la dichotomie, on trouve que la profondeur de l'arbre est de l'ordre de $O(\log n)$.

On en conclut donc que le tri par fusion possède une complexité en $O(n \log n)$, et une complexité spatiale en $O(n)$ puisque la fonction montrée à la sous-sous-section 7.6.2 utilise un stockage intermédiaire pour le résultat dont la taille est au plus celle de la liste de départ.

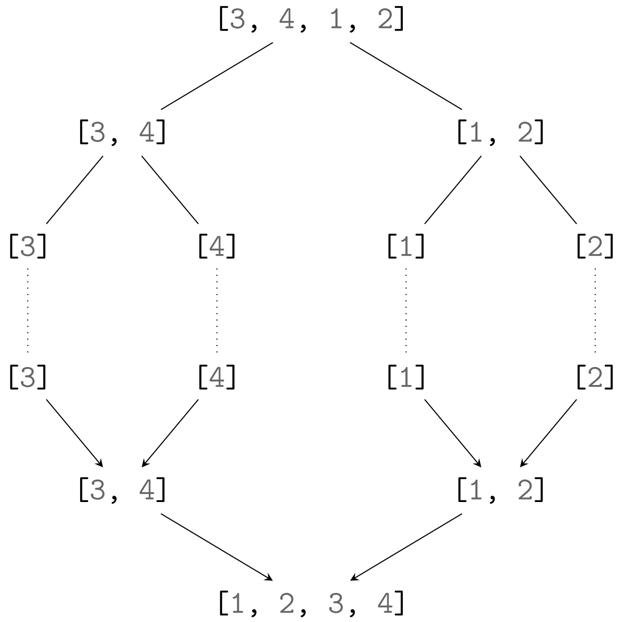
```python
def fusionner(lst, debut, milieu, fin):
    '''Fusionne de façon triée lst[debut:milieu+1] et lst[milieu:fin+1] en temps
    et espace linéaire, en supposant que ces deux sous-listes sont triées.'''
    aux = []
    premier_curseur, second_curseur = debut, milieu + 1
    while premier_curseur <= milieu or second_curseur <= fin:
        # si une des deux sous-listes a été copiée, copier l'autre
        if premier_curseur > milieu:
            aux.extend(lst[second_curseur:fin+1])
            break

        elif second_curseur > fin:
            aux.extend(lst[premier_curseur:milieu+1])
            break

        # sinon, copier min(lst[premier_curseur], lst[second_curseur])
        elif lst[premier_curseur] < lst[second_curseur]:
            aux.append(lst[premier_curseur])
            premier_curseur += 1

        else:
            aux.append(lst[second_curseur])
            second_curseur += 1

    # écraser la section à trier de lst à l'aide de la liste auxiliaire
    lst[debut:fin+1] = aux
```

ALGORITHME 23: Fusion de deux sous-listes triées en une liste triée.

Améliorations du tri fusion

De nombreuses améliorations du tri fusion ont été proposées, qui ne changent pas sa complexité mais améliorent ses performances en pratique. Parmi elles, mentionnons la variante “naturelle” qui consiste à tirer parti des sous-séquences croissantes de la séquence donnée en entrée : l'algorithme effectue une première passe en $O(n)$ pour identifier les sous-séquences croissantes, et utilise la partition résultante pour décider quelles sous-listes fusionner. Ceci évite de découper inutilement des sous-listes déjà triées en morceaux plus petits.

Exemple 41. Voici les étapes de l'algorithme du tri fusion classique sur la liste `[3, 4, 1, 2]` :



La version “naturelle” aurait identifié les sous-séquences croissantes `[3, 4]` et `[1, 2]`, et se serait donc contenté d'une unique fusion triée de ces deux sous-séquences.

7.7 Remarques

La récursivité semble avoir un mauvais bilan, et il serait tentant pour des raisons d'efficacité de tenter de trouver une approche itérative pour résoudre nos problèmes, quitte à produire du code extrêmement complexe. En pratique, Python met toutefois à notre disposition quelques mécanismes qui nous permettent de rentabiliser cette technique.

7.7.1 Paramètres par défaut

Comme dans bien d'autres langages, on peut en Python se passer des fonctions auxiliaires en utilisant des **paramètres par défaut**, c'est-à-dire des paramètres qui prennent une certaine valeur par défaut si l'utilisateur ne précise rien. On consultera la documentation de

Python pour plus d'informations à ce sujet, mais il est important de retenir au moins les deux règles suivantes :

1. ils doivent être placés après tous les autres paramètres "normaux" de la fonction;
2. ils ne peuvent pas recevoir des valeurs par défaut variables ou non encore définies; par exemple, `def ma_fonction(ma_liste, n=len(ma_liste))` est interdit.

Ces paramètres par défaut nous permettent d'éviter la définition de fonctions auxiliaires. Par exemple, on pourrait réécrire la recherche récursive dans une liste comme suit, sans fonction auxiliaire :

```
def recherche_element_2(liste, element, position=None):
    """Renvoie la position de l'élément dans la liste s'il se trouve entre le
    début et la position donnée, ou None dans le cas contraire."""
    # appel initial? -> initialiser position
    if position is None:
        position = len(liste) - 1

    if position < 0:  # condition d'arrêt
        return None

    if liste[position] == element:
        return position

    return recherche_element_2(liste, element, position - 1)
```

7.7.2 Compromis performances / mémoire en Python

On a vu au chapitre 6 des techniques permettant d'utiliser la mémoire pour accélérer les calculs. Python met à disposition le module functools qui contient un mécanisme extrêmement simple permettant, en une ligne, de transformer une fonction récursive en une fonction dont les appels récursifs sont remplacés par des consultations d'un tableau dans lequel on stocke des résultats intermédiaires. Il s'agit d'un "décorateur" nommé `lru_cache`; on n'insistera pas sur cette notion avancée, mais on peut néanmoins en illustrer les effets spectaculaires sur un exemple simple.

Exemple 42. Reprenons l'algorithme 18, qui calcule récursivement les nombres de Fibonacci, et mesurons son temps d'exécution pour le calcul de F_{35} :

```
>>> def fiboR(n):
...     if n < 2:
...         return n
...     return fiboR(n - 1) + fiboR(n - 2)
...
>>> from timeit import timeit
>>> timeit("fiboR(35)", setup="from __main__ import fiboR", number=1)
>>> 2.8152124930056743
```

Utilisons maintenant exactement la même fonction pour refaire ce calcul, mais "décorons"-la cette fois à l'aide de `lru_cache` :

```
>>> from functools import lru_cache
```

```
>>> @lru_cache(maxsize=None)
... def fiboR(n):
...     if n < 2:
...         return n
...     return fiboR(n - 1) + fiboR(n - 2)
...
>>> timeit("fiboR(35)", setup="from __main__ import fiboR", number=1)
>>> 7.915298920124769e-05
```

Le décorateur utilise une table dont la taille n'est pas limitée (d'où le `maxsize=None`) pour stocker les résultats des calculs des appels récursifs. Ainsi, lorsqu'on effectue la décomposition $F_{35} = F_{34} + F_{33}$, les valeurs nécessaires au calcul de F_{34}, dont F_{33}, sont enregistrées dans une table ; et lorsqu'on doit additionner F_{34} à F_{33}, on consulte alors simplement la valeur de F_{33} dans la table au lieu de refaire un appel récursif. Ceci explique le gain phénoménal en temps de calcul que l'on observe en pratique.

--- *(fin exemple 42)* - -

Bibliographie

[1] S. Skiena, *The Algorithm Design Manual*, Springer, 2ème édition, 2008.