Algorithmique avancée – Bloc 5 du DIU " Enseignement de l'Informatique au Lycée " à l'Université de Montpellier

Bruno Grenet

Algorithmique avancée

Bloc 5 du DIU « Enseignement de l'Informatique au Lycée »

Bruno Grenet
Université de Montpellier

Juin 2020

Ce texte présente les principaux points du programme du bloc 5 *Algorithmique avancée* du diplôme inter-universitaire « Enseignement de l'Informatique au Lycée » (*cf.* `https://sourcesup.renater.fr/www/diu-eil/`). Les choix de rédaction ont été effectués dans l'idée de rester relativement proche du programme en vigueur de NSI en terminale générale, tout en dépassant ce programme à plusieurs reprises. En particulier, la partie 3 possède une quatrième partie largement hors programme de NSI, et hors programme du DIU.

Tout retour est bienvenu par email à l'adresse `bruno.grenet@umontpellier.fr`, en particulier pour éliminer des erreurs et typos qui restent certainement [1].

Exercice 0.1. Il est recommandé d'essayer de programmer en Python chacun des algorithmes présentés dans ce texte !

La mise en page de ce document est fortement inspirée du canevas *Legrand Orange Book*, disponible à l'adresse `http://www.latextemplates.com/template/the-legrand-orange-book`.

1. Je remercie vivement Éric Zabban pour sa relecture détaillée de ces notes.

Bibliographie commentée

Les ouvrages suivants couvrent ce qui est présenté dans ce texte, et permettent surtout d'aller (beaucoup) plus loin.

§ **Parties 1 et 2**

— Th. Cormen, Ch. Leiserson, R. Rivest, C. Stein. *Introduction à l'algorithmique*, Dunod, 2004 (traduit de l'américain).

> La fameuse bible d'algorithmique. Ce n'est pas l'ouvrage que je préfère, mais il faut reconnaître que son exhaustivité est impressionnante.

— J. Erickson. *Algorithms*, auto-publié, 2018.

> Mon bouquin préféré d'algorithmique. Très complet, avec de nombreux exercices. Disponible en ligne : `https://algorithms.wtf`.

— S. Dasgupta, Ch. Papadimitriou, U. Vazirani. *Algorithms*, McGraw-Hill, 2006.

> Super bouquin très concis, efficace (l'anti-thèse du « Cormen »). Disponible très facilement [2] en ligne.

— D. Beauquier, J. Berstel, Ph. Chrétienne. *Éléments d'algorithmique*, Masson, 1992.

> Ouvrage épuisé mais disponible en ligne : `http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.html`.

§ **Partie 3**

— E. Asarin. *Calculabilité*, support de cours et de TD, 2003.

> Mon introduction préférée à la calculabilité ! En 44 pages, l'essentiel est présenté. En ligne : `https://www.irif.fr/~asarin/calc2k3/calcul_cours.pdf`.

— S. Perifel. *Complexité algorithmique*, Ellipses, 2014.

> Très bon bouquin sur la théorie de la complexité, avec une approche assez *scolaire*. Disponible en ligne : `https://www.irif.fr/~sperifel/livre_complexite.html`.

— B. Barak. *Introduction to Theoretical Computer Science*, en cours d'écriture.

> Ce bouquin ambitieux, toujours en cours d'écriture (mais qui compte déjà 600 pages), explore de manière assez complète [3] la théorie du calcul, de la calculabilité historique à la complexité la plus moderne. Beaucoup de références à d'autres très bons ouvrages anglophones. En ligne : `https://introtcs.org/`.

2. Je ne mets pas de lien car je doute de leur légalité.
3. Certes, il manque les fonctions récursives primitives, *cf.* 3.4.4.

Table des matières

1 Algorithmes classiques

Cette partie est consacrée à des algorithmes sur les structures de données classiques que sont les arbres binaires et les graphes.

1.1 Arbres binaires

Dans cette partie, on s'intéresse aux algorithmes classiques sur les arbres binaires. On suppose la structure de données connue mais on rappelle dans une première partie le vocabulaire et les notations utiles. On ne se préoccupe pas de l'implantation précise sous-jacente.

1.1.1 Rappel de vocabulaire et notations

Un *arbre binaire* est une structure de données hiérarchique, constituée de *nœuds*. Le nœud initial est appelé *racine*. Chaque nœud peut posséder jusqu'à deux *fils* (fils gauche et droit), dont il est le *père*. La racine est le seul nœud de *niveau* 0, et les fils d'un nœud de niveau i sont des nœuds de niveau $i+1$. Le niveau maximal d'un nœud est appelé la *hauteur* de l'arbre. Un nœud sans fils est une *feuille*. Seule la racine n'a pas de père. Chaque nœud possède une *valeur*, qui peut être n'importe quel objet (entier, chaîne de caractère, ...).

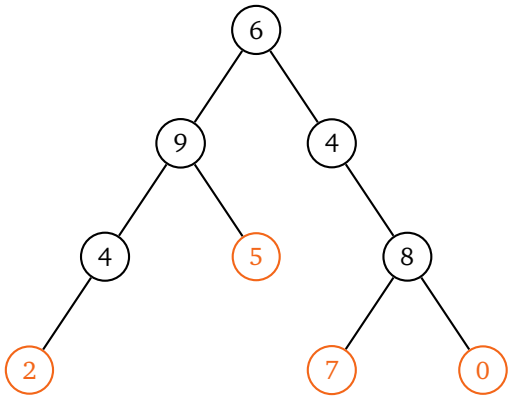


FIGURE 1 – Exemple d'arbre binaire de racine de valeur 6, à 9 nœuds et de hauteur 3. Il possède 4 feuilles (en couleur).

La valeur d'un nœud x est notée $\text{val}(x)$. Ses fils sont notés $\text{filsG}(x)$ et $\text{filsD}(x)$ et son père est noté $\text{pere}(x)$. On utilise le nœud spécial $\emptyset$ (nœud vide) quand $\text{filsG}(x)$ (ou $\text{filsD}(x)$ ou $\text{pere}(x)$) n'existe pas. Si $\text{filsG}(x) \neq \emptyset$, $\text{pere}(\text{filsG}(x)) = x$, et de même pour le fils droit.

Un arbre binaire est une structure récursive. On appelle *descendants* d'un nœud l'ensemble constitué de ses fils, des fils de ses fils, etc. L'arbre dont le fils gauche d'un nœud x est racine est appelé le *sous-arbre gauche* de x. On définit de même le *sous-arbre droit* d'un nœud x.

Exercice 1.1. Montrer que la hauteur h d'un arbre binaire A peut se calculer récursivement de la manière suivante : si la racine est une feuille, la hauteur de A est 0 ; sinon, la hauteur de A est $1 + \max(h_g, h_d)$ où h_g est la hauteur de son sous-arbre gauche et h_d celle de son sous-arbre droit. Par convention, la hauteur d'un arbre dont la racine est le nœud vide est -1 : pourquoi choisir cette convention ?

1.1.2 Algorithmes de base sur les arbres binaires

Les arbres binaires sont une structure de données fondamentalement récursive. Ainsi, les algorithmes sur les arbres binaires sont *a priori* des algorithmes récursifs. On peut baser un grand nombre des algorithmes sur les arbres binaires sur la notion de *parcours* : parcourir un arbre, c'est passer par chacun de ses nœuds. Dans sa version la plus simple, il s'agit simplement d'afficher la valeur de chaque nœud.

Algorithme 1.1 – ParcoursInfixe

Entrée : La racine x d'un arbre binaire A
Sortie : L'algorithme affiche tous les nœuds de A

```
1 Si x ≠ ∅ :
2     ParcoursInfixe(filsG(x))
3     Afficher val(x)
4     ParcoursInfixe(filsD(x))
```

Théorème 1.2 L'algorithme ParcoursInfixe affiche les valeurs de tous les nœuds de A, en temps $O(n)$ où n est le nombre de nœuds dans A.

Démonstration. Pour démontrer que ParcoursInfixe affiche bien les valeurs de tous les nœuds de A, on utilise une *induction structurelle*, c'est-à-dire une récurrence sur la structure des arbres binaires. On effectue la démonstation en détail pour montrer le fonctionnement de cette induction, bien que le résultat en lui-même soit évident.

Le cas de base est l'arbre vide : il n'a aucun nœud et aucun affichage n'est effectué. Si $A \neq \emptyset$, la valeur de sa racine est affichée à la ligne 3. Par hypothèse d'induction, les valeurs de tous les nœuds du sous-arbre gauche de la racine sont affichées par l'appel récursif de la ligne 2. De même, les valeurs des nœuds de son sous-arbre droit sont affichées par l'appel récursif de la ligne 4. Puisque tout nœud est soit la racine, soit appartient à l'un des deux sous-arbres de la racine, les valeurs de tous les nœuds sont affichées par ParcoursInfixe.

La complexité vient du fait que chaque nœud est affiché une fois. On peut également la démontrer par induction structurelle. La complexité pour l'arbre vide est constante. Si on note n_G le nombre de nœuds du sous-arbre gauche et n_D le nombre de nœuds du sous-arbre droit, la complexité de l'appel récursif de la ligne 2 est par hypothèse $O(n_G)$, et celui de la ligne 4 est $O(n_D)$. La complexité totale est donc $O(n_G) + O(n_D) + O(1) = O(n_G + n_D + 1)$. Le nombre total de nœuds dans l'arbre étant $n = n_G + n_D + 1$, la complexité est $O(n)$. □

À côté du parcours infixe, on peut effectuer les parcours *préfixe* et *suffixe*, qui sont très proches. Alors que le parcours infixe affiche d'abord les nœuds du sous-arbre gauche, puis la racine, puis le sous-arbre droit, le parcours préfixe affiche la racine en premier et le parcours suffixe affiche la racine en dernier.

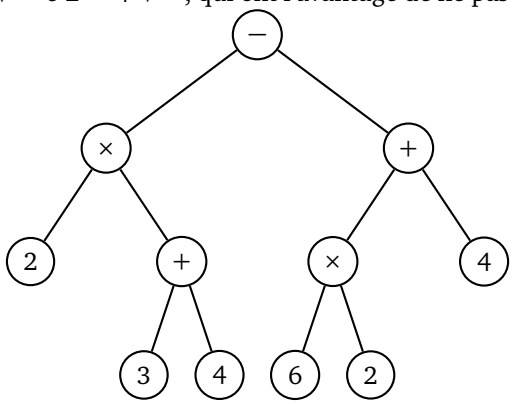
Algorithme 1.3 – PARCOURSPRÉFIXE

```
1 Si x ≠ ∅ :
2     Afficher val(x)
3     PARCOURSPRÉFIXE(filsG(x))
4     PARCOURSPRÉFIXE(filsD(x))
```

Algorithme 1.4 – PARCOURSSUFFIXE

```
1 Si x ≠ ∅ :
2     PARCOURSSUFFIXE(filsG(x))
3     PARCOURSSUFFIXE(filsD(x))
4     Afficher val(x)
```

E On peut représenter une expression arithmétique par un arbre binaire, dont les valeurs des nœuds sont soit des opérations (disons $+$, $-$, $\times$) soit des entiers (pour les feuilles). Par exemple, l'expression $2 \times (3+4) - (6 \times 2 + 4)$ est représentée par l'arbre ci-dessous. Le parcours infixe de l'arbre fournit l'expression (en ajoutant les parenthèses nécessaires). Le parcours préfixe correspond lui à la *notation polonaise* $- \times 2 + 3\ 4 + \times 6\ 2\ 4$ et le parcours suffixe à la *notation polonaise inversée*[a] $2\ 3\ 4 + \times 6\ 2 \times 4 + -$, qui ont l'avantage de ne pas nécessiter de parenthèse !



a. Utilisée sur certaines calculatrices HP par exemple.

Exercice 1.2. Appliquer les trois parcours à l'arbre de la figure 1, page 4.

Étant donné ces parcours, on peut effectuer des traitements sur l'arbre, plutôt que de se contenter d'afficher les valeurs. On prend l'exemple du calcul de la hauteur et du nombre de nœuds d'un arbre binaire. Les deux algorithmes sont quasiment identiques.

Algorithme 1.5 – HAUTEUR

Entrée : La racine x d'un arbre binaire A
Sortie : La hauteur de A

```
1 Si x ≠ ∅ :
2     h_G ← HAUTEUR(filsG(x))
3     h_D ← HAUTEUR(filsD(x))
4     Renvoyer 1 + max(h_G, h_D)
5 Renvoyer −1
```

Algorithme 1.6 – NOMBRENŒUDS

Entrée : La racine x d'un arbre binaire A
Sortie : Le nombre de nœuds dans A

```
1 Si x ≠ ∅ :
2     n_G ← NOMBRENŒUDS(filsG(x))
3     n_D ← NOMBRENŒUDS(filsD(x))
4     Renvoyer 1 + n_G + n_D
5 Renvoyer 0
```

Lemme 1.7 Les algorithmes HAUTEUR et NOMBRENŒUDS sont corrects, et ont comme complexité $O(n)$, où n est le nombre de nœuds de A.

Exercice 1.3.

1. Démontrer le lemme précédent.
2. Écrire des algorithmes pour tester si une valeur apparaît dans un arbre A, calculer la valeur minimale présente dans l'arbre, compter le nombre de feuilles de l'arbre, …

On s'intéresse maintenant à un problème un petit peu plus complexe : on souhaite parcourir l'arbre *en largeur*, c'est-à-dire afficher la racine, puis les nœuds du niveau 1 (les fils de la racine), puis ceux du niveau 2, etc. Par exemple, on souhaite afficher les nœuds de l'arbre de la figure 1 dans l'ordre 6 9 4 4 5 8 2 7 0. Une première méthode consiste à écrire une fonction auxiliaire qui affiche tous les nœuds d'un niveau donné, puis de l'appeler sur chaque niveau.

Exercice 1.4.

1. Modifier PARCOURSINFIXE pour qu'il prenne en entrée supplémentaire une hauteur h et n'affiche que les nœuds de niveau h dans l'arbre.
2. En déduire un algorithme qui parcourt l'arbre en largeur. Analyser sa complexité.

L'algorithme de l'exercice précédent n'est pas efficace. On peut en fait effectuer ce parcours en largeur en temps linéaire. L'idée pour cela est d'utiliser une *file* [4] pour retenir les nœuds à afficher, dans le bon ordre.

Algorithme 1.8 – PARCOURSLARGEUR

Entrée : La racine x d'un arbre binaire A
Sortie : L'algorithme affiche tous les nœuds de A, niveau par niveau et de gauche à droite

```
1 F ← file vide
2 Si x ≠ ∅, l'ajouter à F
3 Tant que F est non vide :
4     y ← défiler un élément de F
5     Afficher val(y)
6     Si filsG(y) ≠ ∅, l'ajouter à F
7     Si filsD(y) ≠ ∅, l'ajouter à F
```

Théorème 1.9 L'algorithme PARCOURSLARGEUR affiche les valeurs de tous les nœuds de A dans l'ordre du parcours en largeur, en temps $O(n)$ où n est le nombre de nœuds dans A.

Démonstration. La complexité ne pose pas de difficulté, puisque la gestion de la file n'est pas coûteux. De même il est assez clair que tous les nœuds de A sont affichés. On va démontrer que l'ordre est bien celui du parcours en largeur. Pour cela, il suffit de montrer que les nœuds sont insérés dans le bon ordre dans la file (puisque dans ce cas, le défilement respectera l'ordre).

4. Une *file* est une structure de données de type « premier arrivé, premier servi » : quand on *défile* un élément d'une file, on obtient l'élément qui avait été inséré le premier dans la file.

On démontre, par récurrence sur les niveaux i de l'arbre, le résultat suivant : tous les nœuds du niveau $(i-1)$ sont insérés dans la file avant ceux du niveau i, et les nœuds du niveau i sont insérés de gauche à droite. Pour $i = 1$, c'est évident puisque la racine, seul nœud de niveau 0, est le premier nœud inséré dans F et que ses deux fils sont insérés dans le bon ordre. Supposons que c'est le cas au rang $i \geq 1$ et soit x un nœud au niveau $(i+1)$, et p son père. Pour que x soit inséré dans F, il faut que p soit défilé de F. Or tous les nœuds de niveau $(i-1)$ se trouvent avant p dans F par hypothèse de récurrence. Cela signifie qu'ils sont défilés avant p également. En particulier, tous les nœuds de niveau i sont donc insérés dans F avant que p soit défilé, donc avant que x soit inséré. Considérons maintenant un deuxième nœud y au niveau $(i+1)$, situé à droite de x. Si x et y ont le même père p, $x = \text{filsG}(p)$ et $y = \text{filsD}(p)$ donc x est bien inséré dans F avant y. Sinon, le père p_x de x se situe à gauche du père p_y de y, tous deux au niveau i. Par hypothèse de récurrence, p_x se situe avant p_y dans F, donc est défilé avant. Ainsi, x est inséré avant y dans F. Le principe de récurrence permet de conclure. □

RÉSUMÉ

- Les algorithmes sur les arbres binaires sont en général basés sur les parcours *en profondeur* (infixe, préfixe ou suffixe) ou *en largeur*.
- Ces algorithmes de parcours ont une complexité $O(n)$ où n est le nombre de nœuds.

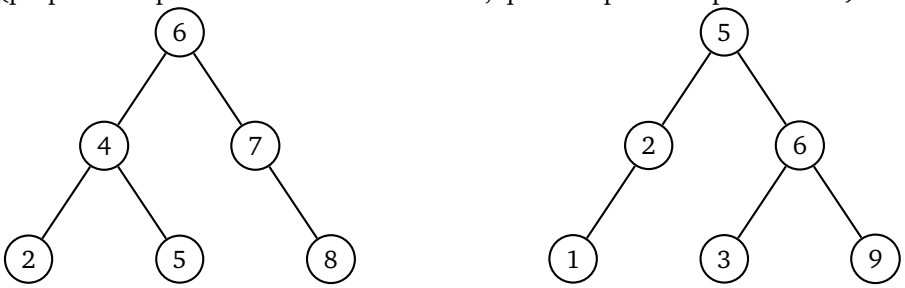
1.1.3 Arbres binaires de recherche

Une utilisation particulière des arbres binaires est la notion d'*arbre binaire de recherche*. On suppose que les valeurs des nœuds sont des objets comparables entre eux : pour fixer les idées, on considère n'avoir que des entiers.

Définition 1.10 Un *arbre binaire de recherche*, ou ABR, est un arbre binaire avec la propriété suivante : pour tout nœud x, tous les nœuds situés dans le sous-arbre gauche de x ont une valeur $\leq \text{val}(x)$, et tous ceux situés dans son sous-arbre droit ont une valeur $\geq \text{val}(x)$.

E

L'arbre de la figure 1 *n'est pas* un ABR car par exemple 9 se situe dans le sous-arbre gauche de 6. Dans la figure ci-dessous, l'arbre de gauche est un ABR, mais celui de droite n'en est pas un car le nœud 3 se situe dans le sous-arbre gauche du nœud 5. On remarque ici qu'être un ABR n'est pas une propriété *locale* : il ne suffit pas que chaque nœud ait son fils gauche inférieur et son fils droit supérieur (propriété respectée dans l'arbre de droite, qui n'est pourtant pas un ABR).



Les ABR servent à maintenir un ensemble de valeurs ordonnées, avec une meilleure complexité qu'une liste chaînée par exemple. On commence par la recherche d'un élément. La structure de l'ABR permet d'éviter de le parcourir en entier.

Algorithme 1.11 – RechercheABR

Entrées : La racine x d'un ABR A, et une valeur v
Sortie : Un nœud de valeur v dans A, ou $\emptyset$ s'il n'en existe pas

```
1 Tant que x ≠ ∅ et val(x) ≠ v :
2       Si val(x) > v : x ← filsG(x)
3       Sinon : x ← filsD(x)
4 Renvoyer x
```

Lemme 1.12 L'algorithme RechercheABR est correct, et sa complexité est $O(h)$ où h est la hauteur de l'ABR.

Démonstration. La correction est relativement évidente, en remarquant que la dernière ligne renvoie bien $\emptyset$ si v n'apparaît pas dans l'ABR. Pour la complexité, on note qu'à chaque passage dans la boucle, le niveau du nœud courant augmente de 1. Le nombre de passages est donc borné par la hauteur de l'arbre. □

La deuxième question qu'on peut se poser est l'ajout d'un nouvel élément dans l'ABR.

Algorithme 1.13 – InsertionABR

Entrées : La racine x d'un ABR A, et une valeur v
Sortie : La valeur v a été insérée dans A

```
1 p ← ∅
2 Tant que x ≠ ∅ :
3       p ← x
4       Si val(x) > v : x ← filsG(x)
5       Sinon : x ← filsD(x)
6 Si p = ∅ : Ajouter un nœud de valeur v comme racine de A
7 Sinon si v < val(p) : Ajouter un nœud de valeur v comme fils gauche de p
8 Sinon : Ajouter un noeud de valeur v comme fils droit de p
```

Lemme 1.14 L'algorithme InsertionABR est correct, c'est-à-dire que si A est un ABR, il reste un ABR après l'algorithme. Sa complexité est $O(h)$ où h est la hauteur de A.

Démonstration. Pour la correction, on note que le nouveau nœud de valeur v se trouve dans le sous-arbre gauche d'un nœud x si et seulement si $\text{val}(x) > v$. Donc A reste bien un ABR s'il en était un avant. La complexité se démontre de la même façon que pour RechercheABR. □

Exercice 1.5.

1. Écrire les deux algorithmes RechercheABR et InsertionABR sous forme récursive.
2. Écrire un algorithme de recherche du minimum dans un ABR, et analyser sa complexité.

On appelle *successeur* d'un nœud x dans A un nœud y tel que $\text{val}(x) \leq \text{val}(y)$ et pour tout nœud $z \neq x, y$, soit

$\text{val}(z) < \text{val}(x)$, soit $\text{val}(z) \geq \text{val}(y)$, et $y = \emptyset$ si x est le nœud de plus grande valeur dans A. *On suppose ici que tous les nœuds ont des valeurs distinctes.*

3. Montrer que le successeur y de x, s'il existe, est le minimum du sous-arbre droit de x.
4. Montrer que le successeur y de x, s'il existe, ne peut pas avoir de fils gauche.
5. En déduire un algorithme de suppression d'un nœud x dans un ABR. *Il faut remplacer x par son successeur, et faire remonter le fils droit de y, s'il existe, à la place de y. Attention au cas où x est l'élément maximal.*

Une caractéristique importante de l'algorithmique des ABR est la dépendance de la complexité en la hauteur de l'arbre, plutôt qu'en son nombre de nœuds. Intuitivement, un ABR sera efficace s'il est *équilibré*, c'est-à-dire de hauteur la plus petite possible.

Théorème 1.15 Soit A un arbre binaire à n nœuds et de hauteur h. Alors

$$h < n < 2^{h+1} \text{ (d'où } \lfloor \log n \rfloor \leq h).$$

Démonstration. Le nombre de nœuds total est égal à la somme du nombre de nœuds contenus à chaque niveau. Si n_i désigne le nombre de nœuds au niveau i, on va montrer que $1 \leq n_i \leq 2^i$. La borne inférieure est évidente. Pour la borne supérieure, on effectue une récurrence sur i. Au niveau $i = 0$, il n'y a que la racine donc $n_0 = 1 = 2^0$. Si $n_i \leq 2^i$, comme chaque nœud du niveau i a au plus deux fils, le nombre de nœuds au niveau $(i+1)$ est $n_{i+1} \leq 2n_i \leq 2^{i+1}$.

On a donc $n = \sum_{i=0}^{h} n_i$. En utilisant la borne précédente, on en déduit que $n \geq \sum_{i=0}^{h} 1 = h+1$, et $n \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$. Une façon facile de montrer cette dernière égalité est de considérer l'écriture binaire des entiers : 2^i est un 1 suivi de i zéros, donc la somme est l'entier constitué de $h+1$ bits égaux à 1 ; c'est bien l'entier $2^{h+1} - 1$.

On en déduit donc que $h < n < 2^{h+1}$. De la deuxième inégalité, en prenant le logarithme [5] des deux côtés, on obtient $\log n < h + 1$. Donc en particulier $h + 1$ est strictement supérieur à la partie entière inférieure de $\log n$, donc $h \geq \lfloor \log n \rfloor$. □

Les opérations sur les ABR ont donc une complexité comprise entre $O(\log n)$ quand ils sont équilibrés, et $O(n)$ s'ils sont particulièrement mal équilibrés.

POUR ALLER PLUS LOIN

L'équilibrage des ABR est un sujet difficile et passionnant. De nombreuses techniques ont été proposées (arbres rouge-noir, arbres AVL, arbres déployés, ...). Parmi celles-ci, une technique est simple à mettre en œuvre : si on insère des éléments dans un ordre aléatoire dans un ABR initialement vide, on obtient avec grande probabilité un ABR équilibré. Puisqu'on n'a pas toujours le choix de l'ordre d'insertion, on peut simuler ce comportement avec la notion de *tarbre*, qui combine la notion d'ABR avec celle de tas (autre structure de données basée sur les arbres binaires).
On peut utiliser les ABR pour effectuer le tri d'un tableau : on insère tous les éléments du tableau dans un ABR initialement vide, puis on supprime itérativement le minimum de l'ABR (jusqu'à obtenir l'ABR vide) pour insérer ces minima successifs dans l'ordre dans le tableau. On peut montrer que cet algorithme est strictement équivalent (en termes de comparaisons effectuées) à l'algorithme du tri rapide. En particulier, en insérant les éléments du tableau dans l'ABR dans un ordre aléatoire, on obtient un ABR de hauteur $O(\log n)$ et l'algorithme a

5. En base 2 comme toujours en informatique !

une complexité $O(n \log n)$.

Résumé

- Les ABR sont des arbres binaires *ordonnés* : le sous-arbre gauche d'un nœud x ne contient que des valeurs $\leq \text{val}(x)$ et son sous-arbre droit que des valeurs $\geq \text{val}\, x$.
- Les opérations courantes sur les ABR (recherche, insertion, suppression) ont une complexité proportionnelle à la hauteur h de l'ABR, qui vérifie $\lfloor \log n \rfloor \leq h < n$ où n est le nombre de nœuds.
- Les techniques d'*équilibrage* des ABR, assez complexes, permettent de garder la hauteur proche de son minimum.

1.2 Graphes

Cette partie est dédiée à l'algorithmique des graphes. On rappelle d'abord quelques notions basiques sur les graphes, le vocabulaire et les notations utiles. On ne s'intéresse que peu à l'implantation sous-jacente. En particulier, les graphes sont classiquement représentés soit par listes d'adjacence, soit par matrice d'adjacence. Les algorithmes décrits dans cette partie seront exprimés de manière assez générique, mais les analyses de complexité prendront en compte ces représentations. On présente dans cette partie les généralisations aux graphes des parcours présentés sur les arbres binaires, ainsi que des utilisations assez directes mais importantes de ces parcours : la recherche de plus courts chemins et la détection de cycles.

1.2.1 Rappel de vocabulaire et notations

Un *graphe* [6] est un ensemble de *sommets* reliés entre eux par des *arêtes*. Deux sommets ne peuvent être reliés que par une arête au maximum. Le *degré* d'un sommet u est son nombre de *voisins*, c'est-à-dire le nombre de sommets v tels qu'il existe une arête entre u et v. Un chemin de longueur ℓ entre deux sommets u et v est un ensemble de sommets $x_0 = u, x_1, \ldots, x_\ell = v$ tels qu'il existe une arête entre x_i et x_{i+1} pour tout $i < \ell$. Un chemin est *simple* si tous ses sommets sont distincts (on ne passe pas deux fois par le même sommet). La *distance* entre deux sommets u et v est la longueur du plus court chemin qui les relie. Un *cycle* est un chemin d'un sommet vers lui-même. Un graphe est *connexe* si deux sommets quelconques sont toujours reliés par (au moins) un chemin.

R On peut voir les arbres binaires comme un cas particulier des graphes. En particulier, un *arbre* est un graphe connexe sans cycle. Un arbre est binaire si chaque sommet est de degré au plus 3. Dans cette vision des arbres binaires, la racine n'est pas spécifiée, et pour un sommet de degré 3, on ne distingue pas le père des fils gauche et droit : c'est une présentation non-hiérarchique des arbres binaires.

La représentation d'un graphe par *listes d'adjacence* est la donnée, pour chaque sommet, de la liste de ses voisins dans le graphe. La *matrice d'adjacence* d'un graphe à n sommets $\{0, \ldots, n-1\}$ est une matrice $n \times n$ dont le coefficient d'indice (u, v) vaut 1 s'il y a une arête $u - v$ dans le graphe, et 0 sinon.

6. On ne parle ici que de graphes *simples* et *non orientés*.

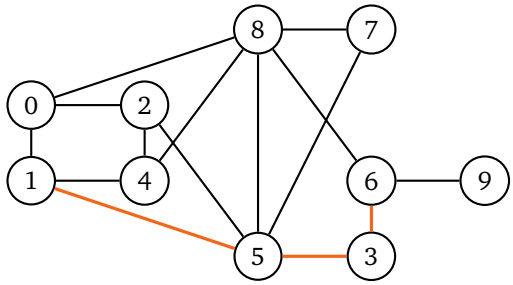


FIGURE 2 – Exemple de graphe à 10 sommets. Le sommet 8 a un degré égal à 5, ses voisins étant 0, 4, 5, 6 et 7. En couleur, un chemin de longueur 3 entre 1 et 6.

1.2.2 Parcours en largeur et plus courts chemins

De même que pour les arbres binaires, on cherche à *parcourir* tous les sommets d'un graphe. On commence par présenter le *parcours en largeur*, qui fait écho au parcours en largeur des arbres binaires. La seule différence est la nécessité de *marquer* les sommets visités pour ne pas les visiter deux fois.

Algorithme 1.16 – ParcoursLargeur

Entrées : Un graphe G et un sommet s de G
Sortie : Affiche tous les sommets du graphe G, si G est connexe

```
1 F ← file vide
2 Ajouter s à F et marquer s
3 Tant que la file F n'est pas vide :
4     u ← défiler F
5     Afficher u
6     Pour tout voisin non marqué v de u :
7         Ajouter v à F et marquer v
```

On n'a pas défini ce qu'on entend par *marquer* un sommet. On peut par exemple supposer que les sommets sont numérotés de 0 à $n-1$. On utilise dans ce cas un tableau de n booléens initialisés à FAUX, et *marquer* un sommet consiste à passer sa case à VRAI.

E Si on applique ParcoursLargeur au graphe de la figure 2 à partir du sommet 0, et en supposant que les voisins d'un sommet sont ajoutés à F dans l'ordre donné par leurs numéros, on obtient les sommets dans l'ordre 0 1 2 8 4 5 6 3 7 9.

Exercice 1.6. Appliquer ParcoursLargeur au graphe de la figure 2 à partir du sommet 9, et à partir du sommet 5.

Théorème 1.17 Si G est connexe, l'algorithme ParcoursLargeur affiche une fois et une seule chaque sommet du graphe, par ordre de distance à s. Sa complexité est $O(m+n)$ si le graphe est représenté par listes d'adjacence, et $O(n^2)$ s'il est représenté par matrice d'adjacence, où m est le nombre d'arêtes et n le nombre de sommets.

Démonstration. Puisque G est connexe, tout sommet $u \neq s$ est à distance finie de s. On montre par récurrence sur d la propriété suivante : tous les sommets de G à distance $\leq d$ de s sont insérés dans F au cours de l'algorithme, et les sommets à distance d de s sont insérés après ceux à distance $d-1$. Pour $d = 1$, c'est clair car le sommet à distance 0, à savoir s, est inséré en premier et ceux à distance 1, les voisins de s, sont insérés ensuite. Supposons que c'est le cas pour une valeur de $d \geq 1$ et considérons un sommet u à distance $(d+1)$. Par définition, il existe un chemin $x_0 = s, x_1, \ldots, x_d, x_{d+1} = u$ dans G (et aucun chemin plus court entre s et u). Puisque x_d est à distance d de s, il est inséré dans F par hypothèse de récurrence, donc u est inséré au plus tard quand x_d est défilé. Gardons la notation x_d pour le sommet à partir duquel u est inséré dans F : x_d ne peut être qu'un sommet à distance d de s. Maintenant, soit v un sommet à distance d de s. Alors v possède un voisin à distance $d-1$ de s, qui par hypothèse a été inséré dans F avant x_d. Donc v est inséré dans F avant u. Le principe de récurrence permet de conclure que tous les sommets à distance finie de s sont insérés dans F au cours de l'algorithme (donc affichés), et que ceux à distance d sont toujours affichés avant ceux à distance $d+1$.

Pour la complexité, on remarque que chaque sommet est inséré une fois et une seule dans la file. Donc la boucle Tant que est exécutée n fois. Ensuite, il faut analyser le coût de la boucle Pour. Si G est représenté par matrice d'adjacence, la boucle Pour doit parcourir les n sommets pour tester s'ils sont voisins de s. On obtient la complexité $O(n^2)$. Si on utilise des listes d'adjacence, on va au total parcourir chaque liste en entier une fois. La somme des degrés des sommets d'un graphe est exactement le double du nombre d'arêtes [7]. Donc le coût cumulé de la boucle Pour à chaque itération de la boucle Tant que est $O(m)$. On en déduit la complexité $O(m+n)$. □

Exercice 1.7. Adapter l'algorithme PARCOURSLARGEUR pour qu'il calcule, pour tout sommet u du graphe, la distance entre s et u. *L'algorithme renvoie un tableau de taille n contenant les distances.*

L'exercice précédent montre qu'on est capable, à l'aide du parcours en largeur, de calculer la *distance* entre un sommet donné et tous les sommets du graphe. L'algorithme de Dijkstra généralise cette approche aux graphes *pondérés* dont chaque arête peut avoir une longueur différente. La longueur d'un chemin est alors définie comme la somme des longueurs des arêtes qui le compose.

Formellement, on se donne une *fonction de longueur* ℓ sur les arêtes du graphe, qui associe à chaque arête une longueur positive ou nulle : $\ell(u,v) = \ell(v,u) \geq 0$ pour tous sommets u et v du graphe reliés par une arête. L'algorithme utilise une *file de priorité* : c'est simplement une structure de données qui permet de représenter un ensemble d'éléments, et d'en extraire rapidement l'élément de *priorité maximale* (*cf.* la démonstration du théorème suivant pour plus de détails).

Algorithme 1.18 – DIJKSTRA

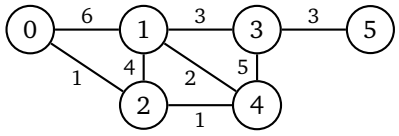
Entrées : Un graphe G, un sommet s de G et une fonction de longueur ℓ sur les arêtes
Sortie : La distance (pondérée par ℓ) entre s et chaque sommet de G

```
1 F ← file de priorité vide
2 D ← tableau de taille n (nombre de sommets du graphe), initialisé à +∞
3 T ← tableau de n booléens, initialisé à FAUX # sommets traités
4 Ajouter s à F
5 D[s] ← 0 # s est à distance nulle de lui-même
6 Tant que la file F n'est pas vide :
```

7. Car chaque arête contribue au degré des deux sommets qu'elle relie.

```
7    u ← sommet non traité de F de distance D[u] minimale
8    T[u] ← VRAI
9    Pour tout voisin v de u :
10       Si D[v] = +∞ : ajouter v à la file F
11       Si T[v] est FAUX : D[v] ← min(D[v], D[u] + ℓ(u, v))
```

E On applique l'algorithme de Dijkstra au graphe suivant, dont les longueurs sont indiqués sur les arêtes, à partir du sommet 0. On affiche la valeur du tableau D au cours de l'algorithme, après le traitement du sommet u (en gras les valeurs modifiées).



u	0	1	2	3	4	5
0	0	**6**	**1**	∞	∞	∞
2	0	**5**	1	∞	**2**	∞
4	0	**4**	1	**7**	2	∞
1	0	4	1	**7**	2	∞
3	0	4	1	7	2	**10**
5	0	4	1	7	2	10

Théorème 1.19 Soit G un graphe connexe, s un sommet de G et ℓ une fonction de longueur. L'algorithme DIJKSTRA calcule les longueurs des plus courts chemins de s à chaque sommet de G. Sa complexité est $O((m+n)\log n)$ si le graphe est représenté par listes d'adjacence, et $O(n^2)$ s'il est représenté par matrice d'ajacence, où m est le nombre d'arêtes et n le nombre de sommets.

Démonstration. La similarité de l'algorithme DIJKSTRA avec PARCOURSLARGEUR permet de calculer facilement sa complexité. La différence avec PARCOURSLARGEUR est la file de priorité : contrairement à une file standard, l'accès au premier élément ne se fait pas en temps $O(1)$. L'implantation la plus classique des files de priorité utilise les *tas*[8] et permet d'accéder au sommet de distance minimale en temps $O(\log n)$, ce qui conduit à la complexité $O((m+n)\log n)$ pour un graphe représenté par listes d'adjacence. On peut également, à la ligne 7, parcourir tous les sommets de F et garder celui qui a la plus courte distance au sommet initial s. Cette stratégie mène à la complexité annoncée pour la matrice d'adjacence.

Pour montrer que l'algorithme est correct, on démontre par récurrence sur le nombre de sommets traités que pour chaque sommet traité v (*i.e.* tel que $T[v] = \text{VRAI}$), $D[v]$ contient la distance entre s et v. On remarque en premier lieu que $D[v]$ ne peut contenir qu'une valeur supérieure ou égale à cette distance. Le premier sommet traité est le sommet initial s, et $D[s] = 0$. Supposons maintenant que k sommets ont été traités et que pour chacun $D[v]$ contient la distance entre s et v. Soit x le $(k+1)^{\text{ème}}$ sommet traité. Il est extrait de F à l'étape 7. On considère un plus court chemin entre s et x, et on veut montrer qu'à l'étape $k+1$, $D[x]$ contient la longueur de ce plus court chemin. Puisque s a déjà été traité mais pas x, il existe une arête $y-z$ dans ce chemin telle que y a déjà été traité, mais z pas encore. Par hypothèse de récurrence, $D[y]$ contient la distance entre s et y et après le traitement de y, $D[z] \leq D[y] + \ell(y,z)$. Or la distance entre s et x est supérieure ou égale à $D[y] + \ell(y,z)$ puisque ces

8. Autre structure de données basée sur les arbres binaires, non présentées ici.

sommets sont sur un plus court chemin entre s et x. D'autre part, le sommet x sélectionné à l'étape $(k+1)$ a la valeur $D[x]$ minimale parmi les sommets à traiter : donc $D[x] \leq D[z] \leq D[y] + \ell(y,z)$. Donc $D[x]$ est inférieure ou égale à la distance entre s et x, et donc égale à cette distance. □

Exercice 1.8. Modifier l'algorithme de Dijkstra pour qu'il renvoie en plus des distances, un plus court chemin de s à u pour tout sommet u de G.

POUR ALLER PLUS LOIN

Avec des files de priorités plus complexes (tas de Fibonacci), on peut atteindre la complexité $O(m + n \log n)$ pour l'algorithme de Dijkstra, voire encore mieux si on sait que les longueurs sont des entiers.
On peut également calculer des plus courts chemins avec des longueurs d'arêtes pouvant être négatives, tant qu'il n'existe pas de cycle de longueur totale strictement négative : en effet, dans un tel cas le *meilleur* chemin consisterait à répéter un infinité de fois le cycle négatif, pour une longueur totale $-\infty$. L'algorithme de DIJKSTRA devient alors exponentiel, mais il existe un autre algorithme, dû à Bellman et Ford, de complexité polynomiale. On peut remarquer que DIJKSTRA est un algorithme *glouton* pour ce problème, alors que l'algorithme de Bellman-Ford est un algorithme de programmation dynamique pour le même problème (*cf.* 2.1).

RÉSUMÉ

- Le parcours en largeur permet de parcourir un graphe, en temps $O(m+n)$ si le graphe est représenté par listes d'adjacence.
- On peut l'étendre au cas de graphes pondérés grâce à l'algorithme de Dijkstra, de complexité $O(m \log n + n)$, qui permet de calculer des plus courts chemins dans un graphe.

1.2.3 Parcours en profondeur et cycles

L'équivalent pour les graphes des parcours infixe, préfixe et suffixe est le parcours en profondeur. Contrairement au cas des arbres binaires, il est nécessaire de retenir, à l'instar du parcours en largeur, une liste de sommets à visiter. L'aspect à remarquer est la très forte similarité du prochain algorithme avec PARCOURSLARGEUR : la seule différence est l'utilisation d'une *pile* (« premier arrivé dernier servi ») à la place d'une *file* (« premier arrivé premier servi »).

Algorithme 1.20 – PARCOURSPROFONDEUR

Entrées Un graphe G et un sommet s de G
Sortie : Affiche tous les sommets du graphe G, si G est connexe

```
1 P ← pile vide
2 Ajouter s à P et marquer s
3 Tant que la pile P n'est pas vide :
4     u ← dépiler P
5     Afficher u
6     Pour tout voisin non marqué v de u :
7         Ajouter v à P et marquer v
```

E Le parcours en profondeur appliqué sur le graphe de la figure 2 à partir du sommet 0 produit l'ordre 0 8 7 6 9 3 5 4 2 1 si on ajoute toujours les sommets dans l'ordre de leurs numéros.

Exercice 1.9.

1. Appliquer le parcours en profondeur au graphe de la figure 2 à partir du sommet 9 et à partir du sommet 5.
2. Écrire une version récursive de ParcoursProfondeur, sans utiliser de pile.

Théorème 1.21 Si G est connexe, l'algorithme ParcoursProfondeur affiche une fois et une seule chaque sommet du graphe. Sa complexité est $O(m+n)$ si le graphe est représenté par listes d'adjacence, et $O(n^2)$ s'il est représenté par matrice d'ajacence, où m est le nombre d'arêtes et n le nombre de sommets.

La preuve de ce théorème est similaire à celle pour ParcoursLargeur, en plus simple car on ne garantit rien sur l'ordre des sommets. Le parcours en profondeur peut être utilisé pour détecter la présence d'un cycle dans un graphe (connexe), autrement dit déterminer s'il s'agit ou non d'un arbre. L'idée est de modifier légèrement le parcours en profondeur grâce à la remarque suivante : s'il n'existe pas de cycle, il existe un seul chemin entre le sommet initial et n'importe quel sommet du graphe ; à l'inverse, un cycle assure qu'il existe au moins un sommet qui est atteignable depuis le sommet initial par au moins deux chemins distincts. Au lieu de *marquer* les sommets visités avec un booléen Vrai/Faux, on les marque avec le numéro de leur *prédécesseur* dans un chemin depuis le sommet initial. On détecte alors un cycle si un sommet a deux prédécesseurs possibles.

Algorithme 1.22 – DétectionCycle

Entrée : Un graphe connexe G sur les n sommets $\{0,\dots,n-1\}$
Sortie : Existe-t-il un cycle dans G ?

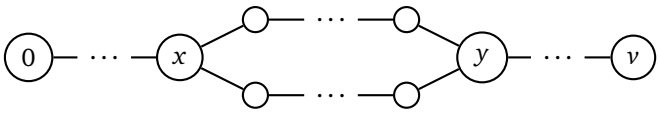
```
1  Pred ← tableau de n entiers, initialisé à −1 # Prédécesseurs
2  P ← pile vide
3  Pred[0] ← 0
4  Ajouter 0 à P
5  Tant que la pile P n'est pas vide :
6      u ← dépiler P
7      Pour tout voisin v de u :
8          Si Pred[v] = −1 :
9              Pred[v] ← u
10             Ajouter v à P
u.         Sinon si v ≠ Pred[u] :
v.             Renvoyer Vrai
w. Renvoyer Faux
```

Théorème 1.23 Si G est connexe, DétectionCycle renvoie Vrai si et seulement si G possède un cycle, en temps $O(n)$ si le graphe est représenté par listes d'adjacence et $O(n^2)$ s'il est représenté par matrice d'adjacence.

Démonstration. On remarque une différence de complexité par rapport au parcours en profondeur. En effet, dans les parcours on *visite* une fois chaque arête. Pour la détection de cycle, on s'arrête dès qu'on trouve un cycle. Or on peut montrer facilement qu'un graphe qui possède $\geq n$ arêtes possède

forcément un cycle : ainsi, au moment où l'algorithme s'arrête, il a visité au plus n arêtes. Cela justifie la complexité dans le cas de listes d'adjacence, mais ne change pas la complexité dans le cas des matrices d'adjacence.

Montrons en premier lieu qu'il y a un cycle dans G si et seulement s'il existe un sommet $v \neq 0$ tel qu'il existe deux chemins distincts entre 0 et v. S'il existe deux chemins distincts, ils peuvent partager un début commun et une fin commune. On note x le dernier sommet avant v qu'ils ont en commun en partant de 0 et y le dernier qu'ils ont en commun en partant de v. Alors les deux chemins distincts entre x et y forment un cycle (*cf* dessin ci-dessous). De même, s'il existe un cycle, on considère n'importe quel sommet y du cycle et un chemin entre 0 et y. On note x le premier sommet du cycle qui apparaît sur le chemin de 0 vers y. Si $x \neq y$, on a trouvé deux chemins entre 0 et y. Sinon, on considère n'importe quel *autre* sommet v du cycle. Alors on a deux chemins entre 0 et v : ils partagent le même début (entre 0 et y) puis empruntent le cycle dans les deux directions possibles.



Ainsi, s'il n'existe pas de cycle dans G, tout sommet v n'est atteignable depuis 0 que par un seul chemin : dans l'exploration du parcours en profondeur, v ne peut avoir qu'un seul précédesseur, ses autres voisins ne peuvent être visités qu'en provenant de v. Réciproquement, s'il existe un cycle dans G, on peut considérer un sommet x comme dans le dessin précédent qui est le premier sommet du cycle à être visité. Depuis x, le parcours en profondeur empile tous les sommets du cycle (disons en commençant par le chemin *du bas*, en passant par y et en finissant par le chemin *du haut*). Lorsque le premier sommet du chemin du haut (voisin de x) est dépilé, appelons le s, x a déjà été visité mais son prédécesseur n'est pas s : le cycle est detecté. □

Résumé

- Le parcours en profondeur permet de parcourir un graphe, en temps $O(m+n)$ si le graphe est représenté par listes d'adjacence.
- On peut l'utiliser pour détecter la présence d'un cycle dans un graphe.

2 Algorithmes avancés

Cette partie est consacrée à des sujets plus avancés d'algorithmique. On commence par la *programmation dynamique*, qui est un paradigme algorithmique. À ce titre, c'est la suite logique du cours sur *diviser pour régner* et les *algorithmes gloutons*. Ensuite, on présente le problème de la recherche de motif dans un texte, et l'un des algorithmes les plus efficaces pour ce problèmes, à savoir l'algorithme de Boyer-Moore.

2.1 Programmation dynamique

La *programmation dynamique* est un paradigme algorithmique, au même titre que *diviser pour régner* ou les *algorithmes gloutons*. C'est donc une méthode algorithmique, applicables à de nombreux problèmes. Pour la présenter, on commence par étudier en détail un problème particulier, le rendu de monnaie, avant de dégager les caractéristiques principales et les enjeux de cette méthode. On

présente ensuite un deuxième exemple qui utilise la programmation dynamique de manière un peu plus complexe, l'alignement de séquences.

2.1.1 Un premier exemple : le rendu de monnaie

Le problème du *rendu de monnaie* consiste à déterminer le nombre minimal de billets et pièces nécessaires pour rendre une somme donnée. L'entrée est constituée de l'ensemble des valeurs des pièces [9] et de la somme à rendre. La sortie est soit une liste de pièces, minimale pour atteindre cette somme, soit simplement le nombre minimal de pièces nécessaires (sans avoir la liste). On le définit formellement.

Problème 2.1 – RenduDeMonnaie

Entrées : Ensemble P d'entiers, entier s
Sortie 1 : Nombre minimal de pièces de P dont la somme vaut s
Sortie 2 : Liste de pièces de P, minimale, dont la somme vaut s

E Le système des euros avec des pièces à partir de 1 centime et des billets jusqu'à 500€ correspond à l'entrée $\{50000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5, 2, 1\}$. Si on doit rendre 14€53, la meilleure solution est de rendre les 6 pièces 10€ + 2 × 2€ + 50 cent. + 2 cent. + 1 cent. La sortie 1 est donc 6 et la sortie 2 est la liste $[1000, 200, 200, 50, 2, 1]$ dont la somme vaut bien 1453.

La majorité des monnaies fiduciaires sont *canoniques*, c'est-à-dire que l'*algorithme glouton* est optimal pour rendre la monnaie : on commence par rendre la plus grande pièce inférieure ou égale à la somme visée, puis on recommence jusqu'à arriver à 0. Cependant, on peut facilement créer des monnaies non canoniques [10], comme $\{6, 4, 1\}$ par exemple : pour rendre 8, l'algorithme glouton choisit 6, puis deux fois 1 alors que rendre deux fois 4 est une meilleure solution. On a donc besoin d'un autre algorithme dans le cas général.

Exercice 2.1. Écrire l'algorithme glouton du rendu de monnaie.

§ **Formule récursive** On cherche à déterminer, pour un ensemble P de pièces et une somme s, le nombre minimal de pièces de P pour atteindre la somme s. On note $\text{pieces}_P(s)$ ce minimum. Pour construire un algorithme, on décrit une formule récursive pour cette valeur. Si $s = 0$, alors $\text{pieces}_P(s) = 0$ de manière évidente. Sinon, on peut rendre n'importe laquelle des pièces p de P si $p \leq s$. Si on a sélectionné p, il reste la somme $s - p$ à rendre : cette somme nécessite par hypothèse $\text{pieces}_P(s-p)$ pièces. On en déduit que

$$\text{pieces}_P(s) = \begin{cases} 0 & \text{si } s = 0, \text{ et} \\ 1 + \min\{\text{pieces}_P(s-p) : p \in P, p \leq s\} & \text{sinon.} \end{cases}$$

De cette formule récursive, on tire aisément un algorithme récursif. Il suffit, pour calculer $\text{pieces}_P(s)$, d'effectuer une boucle qui fait un appel récursif pour chaque pièce $p \leq s$.

9. On utilise le mot pièce pour « pièce ou billet ».
10. C'était le cas de la livre sterling avant 1971.

Algorithme 2.2 – RenduNaif

Entrées : Ensemble P d'entiers, entier s
Sortie : Nombre minimal de pièces de P dont la somme vaut s

```
1 Si s = 0 : renvoyer 0
2 n ← +∞
3 Pour chaque pièce p ∈ P :
4   Si p ≤ s :
5     n_p ← RenduNaif(P, s − p)
6     Si n_p ≤ n : n ← n_p
7 Renvoyer 1 + n
```

Exercice 2.2. Modifier l'algorithme précédent pour qu'il renvoie la liste des pièces, plutôt que leur nombre.

L'algorithme précédent est correct. Mais on ne peut pas vraiment dire qu'il fonctionne… Par exemple avec une implantation en Python, RenduNaif($\{50, 20, 10, 5, 2, 1\}, 30$) met environ cinq secondes à répondre sur un ordinateur standard, et plus d'une minute si on remplace 30 par 35. La raison est le très grand nombre d'appels récursifs. La plupart sont d'ailleurs parfaitement inutiles. Pour s'en rendre compte, on applique l'algorithme sur un exemple pourtant très simple : RenduNaif($\{6, 4, 1\}, 6$). Les appels récursifs sont indiqués en figure 3. On remarque en particulier que plusieurs appels récursifs sont effectués sur les mêmes valeurs ($s = 2$ par exemple), ce qui est une perte de temps évidente.

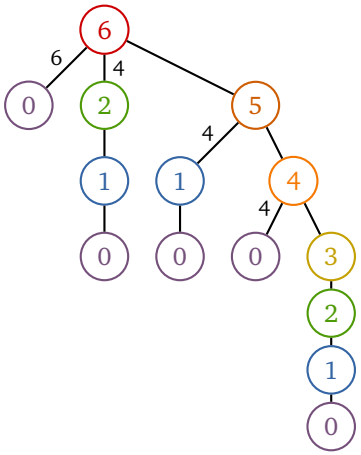


Figure 3 – Les valeurs de s sont représentées sur les sommets, et le choix de pièces sur les arêtes (sauf 1, qui est omise).

§ **Programmation dynamique** Pour régler cette difficulté, une première technique, appelée *mémoïsation*, consiste à retenir l'ensemble des appels récursifs effectués, pour éviter de les refaire quand c'est inutile. Dans notre exemple, il suffit de se rappeler l'ensemble des valeurs s pour lesquelles l'appel récursif a déjà fourni une réponse (ainsi que la réponse). Cette technique est plutôt du domaine de

la programmation que de l'algorithmique, et peut s'automatiser jusqu'à une certaine mesure. On obtient l'algorithme suivant, qui est une adaptation de RenduNaif. Les appels à RenduMemo avec les pièces $\{50, 20, 10, 5, 2, 1\}$ et la somme 30 ou 35 prennent maintenant moins d'une milliseconde.

Algorithme 2.3 – RenduMemo

Entrées : Ensemble P d'entiers, entier s, dictionnaire V associant à des sommes s un nombre V_s de pièces. *On suppose que V contient $V_0 = 0$.*
Sortie : Nombre minimal de pièces de P dont la somme vaut s

```
1 Si s est dans V : renvoyer V_s
2 n ← +∞
3 Pour chaque pièce p ∈ P :
4   Si p ≤ s :
5     n_p ← RenduMemo(P, s − p, V)
6     Si n_p ≤ n : n ← n_p
7 V_s ← 1 + n
8 Renvoyer 1 + n
```

L'idée de la programmation dynamique est très proche de la mémoïsation. Mais au lieu d'effectuer des appels récursifs en partant de la somme s à rendre et de mémoriser les appels récursifs effectués, on calcule explicitement le dictionnaire V, en commençant *par le bas*. Cela revient, une fois qu'on déroule l'algorithme, quasiment à la même chose que RenduMemo. Mais le code est plus clair, et il y a un petit gain car il n'est pas nécessaire de stocker une pile d'appels récursifs [11].

Algorithme 2.4 – RenduProgdyn

Entrées : Ensemble P d'entiers, entier s
Sortie : Nombre minimal de pièces de P dont la somme vaut s

```
1 L ← tableau de longueur s + 1, initialisé à +∞
2 L_0 ← 0
3 Pour i = 1 à s :
4   Pour chaque pièce p ∈ P :
5     Si p ≤ i et L_{i−p} + 1 < L_i :
6       L_i ← L_{i−p} + 1
7 Renvoyer L_s
```

Théorème 2.5 L'algorithme RenduProgdyn calcule $\text{pieces}_P(s)$ en temps $O(s|P|)$ où $|P|$ désigne le nombre de pièces dans P.

Démonstration. La complexité en temps est assez directe : l'algorithme consiste en deux boucles imbriquées, l'une de taille s, l'autre de taille $|P|$.

11. On peut remarquer que RenduMemo$(\{50, 20, 10, 5, 2, 1\}, 1000000, \{0 : 0\})$ provoque une erreur en Python car trop d'appels récursifs sont effectués.

Pour la correction [12], il faut montrer qu'après l'itération i de la boucle, L_i contient $\text{pieces}_P(i)$: ainsi à la fin de l'algorithme, L_s contient $\text{pieces}_P(s)$. La démonstration se fait par récurrence sur i. Avant la première étape, c'est-à-dire *à la fin de l'itération* 0, L_0 contient $0 = \text{pieces}_P(0)$. Si le résultat est correct jusqu'au rang $i-1$, alors après l'itération i, L_i contient $1 + \min\{\text{pieces}_P(i-p) : p \in P, p \leq i\}$ puisque la boucle calcule exactement ce minimum. Or c'est exactement la définition de $\text{pieces}_P(i)$. Le principe de récurrence suffit à conclure. □

R La complexité de cet algorithme *n'est pas* polynomiale en la taille de l'entrée car la taille de s est $\log(s)$ (nombre approximatif de bits nécessaire pour écrire l'entier s). D'autre part, la complexité *en espace* de l'algorithme est $O(s)$ car on stocke l'ensemble du tableau L.

§ **Reconstruction** Le problème du rendu de monnaie admet deux sorties possibles. On a montré comment calculer le nombre de pièces minimum, mais pas encore comment calculer la liste des pièces. On peut bien sûr adapter les algorithmes précédents pour qu'ils renvoient cette liste de pièce. Mais il y a plus simple, en se basant sur l'algorithme de programmation dynamique. Étant donné le tableau L rempli à la fin de RenduProgdyn, on peut calculer cette liste de pièces. Pour cela, on part de la somme s, et on retient la valeur n de L_s. D'après la façon dont est rempli L, il existe une pièce p telle que L_{i-p} contient $n-1$. On cherche cette pièce p et on continue jusqu'à arriver à L_0. L'algorithme suivant implante cette idée. Son troisième paramètre est le tableau L rempli dans RenduProgdyn.

Algorithme 2.6 – RenduReconstruction

Entrées : Ensemble P d'entiers, entier s, tableau L d'entiers
Sortie : Liste de pièces de P, minimale, dont la somme vaut s

```
1 S ← liste vide
2 Tant que s > 0 :
3   Pour chaque pièce p ∈ P :
4     Si p ≤ s et L_{s-p} = L_s − 1 :
5       s ← s − p
6       Ajouter p à S
7       Sortir de la boucle Pour
8 Renvoyer S
```

Lemme 2.7 Si L est le tableau rempli par RenduProgdyn(P,s), RenduReconstruction(P,s,L) renvoie une liste de pièces minimales en temps $O(n|P|)$, où $n = \text{pieces}_P(s)$.

La démonstration de ce lemme, facile, est laissée en exercice.

R Si on souhaite calculer la liste de pièces grâce à RenduReconstruction, il faut modifier RenduProgdyn pour qu'il renvoie le tableau L plutôt que simplement L_s.

12. C'est-à-dire la preuve que l'algorithme calcule effectivement ce que l'on souhaite qu'il calcule.

Résumé

- Le rendu de monnaie consiste à rendre une somme s à l'aide d'une liste de pièces P.
- Pour les monnaies *canoniques*, l'algorithme glouton suffit. Mais pour d'autres monnaies, il ne trouve pas toujours la solution optimale.
- Pour obtenir une solution optimale quelle que soit la monnaie, on décrit une *formule récursive* pour le nombre minimal de pièces.
- L'algorithme obtenu directement depuis cette formule est extrêmement lent. La *programmation dynamique* (ainsi que la *mémoïsation*) permettent d'obtenir un algorithme efficace.
- Pour obtenir la liste des pièces, on *reconstruit* la solution *a posteriori*.

2.1.2 La programmation dynamique, *qu'es aquó* ?

La programmation dynamique est un paradigme algorithmique employé pour résoudre des problèmes d'*optimisation* : trouver le plus petit nombre de pièces par exemple, et de manière plus générale maximiser ou minimiser une quantité donnée. L'idée centrale est d'effectuer de la *récursion sans répétition*. Développer un algorithme de programmation dynamique repose sur trois ingrédients.

1. Obtenir une *formule récursive* pour la valeur optimale.

   On exprime la valeur optimale en fonction de *sous-problèmes*, qui peuvent être nombreux et non disjoints. Cela nécessite une spécification précise du problème. Et il faut être vigilant au fait que la formule récursive doit se baser sur des solutions d'instances plus petites du même problème exactement. C'est le cœur de la technique, parfois difficile à obtenir.

2. Décrire un *algorithme itératif* pour la valeur optimale.

   On se base sur la formule récursive. L'algorithme itératif part des plus petits sous-problèmes et calcule la valeur optimale pour des sous-problèmes de taille croissante, jusqu'au problème d'origine. Cette étape nécessite de réfléchir à l'utilisation d'une structure de données (*très* souvent un tableau) et à la façon de le remplir. On peut alors écrire effectivement l'algorithme et analyser sa complexité. Cette étape est relativement facile lorsqu'on a la formule récursive.

3. *Reconstruire* une solution optimale *a posteriori*.

   Les deux premières étapes permettent de calculer la valeur d'une solution optimale. Cette dernière étape, parfois ignorée car inutile, cherche à expliciter une solution qui atteint la valeur optimale. Pour cela, on doit souvent légèrement modifier l'algorithme itératif de l'étape précédente pour qu'il renvoie un peu plus d'information. L'algorithme de reconstruction *indépendant* se base sur cette information et construit une solution. Alors que l'algorithme itératif va des instances petites vers les plus grandes, l'algorithme de reconstruction part des instances les plus grandes et *redescend* vers les plus petites.

E Dans le problème du RenduDeMonnaie, la formule récursive est l'expression de $\text{pieces}_P(s)$ à partir de $\text{pieces}_P(s-p)$ pour $p \in P$. On en déduit l'algorithme itératif RenduProgdyn. Pour la reconstruction, on modifie RenduProgdyn pour qu'il renvoie le tableau L et on écrit l'algorithme indépendant RenduReconstruction.

§ **Comparaison avec d'autres techniques** La programmation dynamique utilise une approche ascendante pour calculer la valeur optimale, et la formule récursive se base sur de nombreux sous-problèmes non disjoints. En comparaison, le paradigme *diviser pour régner* utilise une approche descendante, et exprime une solution en fonction d'un petit nombre de problèmes disjoints. On peut voir la programmation dynamique comme une extension des algorithme gloutons : ceux-ci sont souvent insuffisants, c'est-à-dire qu'ils ne fournissent que rarement une solution optimale, et la programmation dynamique permet alors dans de nombreux cas de résoudre optimalement le problème.

La *mémoïsation* est un cousin proche de la programmation dynamique, qui conserve l'approche naturelle descendante donnée par la formule récursive mais utilise une mémoire supplémentaire pour éviter les appels récursifs inutiles. C'est plutôt une technique de programmation, qui peut être mise en pratique automatiquement dans des compilateurs, pour améliorer des programmes. Si cette technique est assez simple conceptuellement, elle résulte en des algorithmes moins élégants et souvent (légèrement) moins efficaces que la programmation dynamique.

§ **La problématique de la mémoire** Dans le RENDUDEMONNAIE, on a vu que l'espace mémoire nécessaire est $O(s)$. Pour des grandes valeurs de s, cet espace peut devenir prohibitif. De manière générale en programmation dynamique, la question de l'espace mémoire peut devenir problématique. En particulier lorsqu'on utilise des tableaux multi-dimensionnels (comme dans l'exemple suivant), l'espace mémoire peut devenir le *réactif limitant* de l'algorithme de programmation dynamique.

Dans certains problèmes, il est possible de limiter l'espace mémoire en ne retenant qu'une partie des données. Par exemple dans le cas du rendu de monnaie, si la pièce la plus grande a pour valeur $p_{\max}$, on peut ne retenir que les $p_{\max}$ dernières cases du tableau. En effet, à l'itération i et aux suivantes, on n'accède qu'aux cases à partir de l'indice $i - p_{\max}$. On remarque cependant que dans cet exemple, ne retenir qu'une partie du tableau L empêche ensuite la reconstruction. C'est en fait un phénomène assez général : il est souvent possible de limiter l'utilisation de la mémoire, mais uniquement pour calculer la valeur optimale et non la solution optimale.

R Dans le cas de la mémoïsation, la question de la mémoire se pose dans les mêmes termes. Il est même en général impossible de limiter l'utilisation de la mémoire comme indiqué pour la programmation dynamique, ce qui rend cette dernière préférable dans bien des cas.

§ **D'où vient ce nom ?** Comme déjà indiqué, la programmation dynamique est un paradigme *algorithmique*. Alors pourquoi *programmation*, et pourquoi *dynamique* ? Cette appellation est due à Bellman (1940) qui effectuait des travaux en optimisation mathématique. Le terme *programmation* est utilisé dans son sens de planification ou d'ordonnancement de tâches, et renvoie à la *programmation linéaire*. Quant à *dynamique*, Bellman a déclaré bien des années plus tard dans son autobiographie (1984) : « *it's impossible to use the word dynamic in a pejorative sense. [...] Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to.* ». Selon lui, il avait besoin d'un tel qualificatif pour que l'armée de l'air américaine, qui était l'un de ses principaux financeurs, accepte de continuer de le financer malgré le caractère très théorique et mathématique de ses recherches. Cette histoire est très belle, mais peut-être un peu trop, *cf.* `https://en.wikipedia.org/wiki/Dynamic_programming#History`.

RÉSUMÉ
- La programmation dynamique s'appuie sur trois ingrédients : une *formule récursive*, un *algorithme itératif*, et un éventuel *algorithme de reconstruction*.
- C'est une approche ascendante, qui étend et améliore dans de nombreux cas l'approche

gloutonne, en exprimant le problème à résoudre en fonction de nombreux sous-problèmes non disjoints.
— Une des problématiques importantes est l'utilisation gourmande de la mémoire.

2.1.3 L'alignement de séquences

Pour mesurer la *distance* entre deux chaînes de caractères (qui peuvent être du texte en langue naturelle, de l'ADN, une séquence de protéines, ...), différentes notions de *distance* peuvent être définies. La *distance de Hamming* compte simplement le nombre de caractères distincts. Par exemple, `MANGER` et `MENTIR` sont à distance 3 pour la distance de Hamming (`A` ↛ `E`, `T` ↛ `G`, `E` ↛ `I`).

Dans le problème de l'*alignement de séquences*, on utilise une notion de distance un peu plus évoluée, appelée *distance d'édition*, ou *distance de Levenshtein* ou encore *distance d'Ulam*. À partir de deux mots, on construit un *alignement* des deux mots en insérant des *blancs* (notés `_` et qu'on suppose ne pas apparaître dans les mots d'origine) dans chacun des mots, avec la contrainte que les mots augmentés de blancs doivent avoir la même longueur. La *distance d'édition* entre deux mots u et v est définie comme le nombre minimal de *désaccords* dans un alignement de u et v, c'est-à-dire la distance de Hamming minimale d'un alignement de u et v.

E Étant donné les deux mots `PECHEURS` et `ECRITURE`, on peut construire l'alignement

```
PECHE_URS
_ECRITURE
```

en insérant un blanc dans chacun des deux mots. Dans cet alignement, il y a 5 désaccords : le `P` est aligné avec un blanc `_`, le `H` avec un `R`, le ($2^{\text{ème}}$) `E` avec un `I`, le blanc `_` inséré *en haut* avec un `T`, et le `S` avec un `E`. Les autres lettres sont en accord (le `C` est aligné avec un `C`, etc.). La distance d'édition est donc ≤ 5. On peut vérifier que la distance d'édition est exactement 5 dans ce cas.

Exercice 2.3. Calculer la distance d'édition entre `ALGORITHME` et `POLYRYTHMIE`, ou entre `NICHE` et `CHIEN`.

Le problème de l'alignement de séquences consiste à calculer un alignement optimal de deux mots u et v, ou du moins à calculer la distance d'édition entre u et v.

Problème 2.8 – Alignement

Entrées : Deux mots u et v (de tailles quelconques)
Sortie 1 : La distance d'édition entre u et v
Sortie 2 : Un alignement de u et v, avec le nombre minimal de désaccords

On peut tenter d'écrire un algorithme directement pour ce problème, mais même l'algorithme de force brute (tester toutes les possibilités) n'est pas si évident que ça. On va recourir à la programmation dynamique.

R On peut aussi définir la distance d'édition comme le nombre minimal d'*opérations élémentaires* pour passer d'un mot u à un mot v, où les opérations élémentaires autorisées sont : l'insertion d'une lettre, la suppression d'une lettre, et la substitution d'une lettre par une autre. Par exemple, on retrouve que `PECHEURS` et `ECRITURE` sont à distance 5 :

`PECHEURS` → `ECHEURS` → `ECREURS` → `ECRIURS` → `ECRITURS` → `ECRITURE`.

L'alignement ou la séquence d'opérations sont tout à fait équivalentes pour définir la distance d'édition : un désaccord `L/_` correspond à une suppression de `L`, `_/L` à une insertion, et `L/M` à une substitution.

§ **Formule récursive** On s'intéresse pour l'instant uniquement au calcul de la distance d'édition. Pour définir la formule récursive, on utilise la définition à l'aide de l'alignement, en supposant qu'on aligne progressivement les deux mots. Pour cela, on note u_i la $i^{\text{ème}}$ lettre de u et $u_{[0,i[}$ le préfixe de u constitué des lettres d'indices 0 à $i-1$, et de même pour v. Par exemple si u est le mot `PECHEURS`, $u_3 =$ `H` et $u_{[0,5[} =$ `PECHE`. On note également m la longueur de u et n celle de v, de telle sorte que $u_{[0,m[} = u$ et $v_{[0,n[} = v$.

On cherche à aligner deux mots, comme dans l'exemple `PECHE_URS/_ECRITURE`. On s'intéresse à la dernière lettre dans l'alignement. Il y a trois cas possibles : c'est soit une insertion (ex. `_/E`), soit une suppression (ex. `S/_`), soit une substitution (ex. `S/E`). Dans le premier cas, cela signifie qu'on a aligné u avec $v_{[0,n-1[}$ puis un blanc avec la dernière lettre de v. Dans le deuxième, on a aligné $u_{[0,m-1[}$ avec v et la dernière lettre de u avec un blanc. Dans le troisième, on a aligné $u_{[0,m-1[}$ avec $v_{[0,n-1[}$ et effectué la substitution $u_{m-1} \to v_{n-1}$. Un des trois cas doit se produire, et celui qui fournit la distance la plus petite est le meilleur. La distance entre u et v est la distance entre les deux préfixes, plus éventuellement 1 pour la dernière lettre de l'alignement.

Notons $\text{edit}(i,j)$ la distance entre $u_{[0,i[}$ et $v_{[0,j[}$, de telle sorte qu'on cherche $\text{edit}(m,n)$. Alors la discussion précédente revient à dire que

$$\text{edit}(m,n) = \min \begin{cases} \text{edit}(m,n-1)+1 \\ \text{edit}(m-1,n)+1 \\ \text{edit}(m-1,n-1)+\mathbf{1}_{[u_{m-1}\neq v_{n-1}]} \end{cases}$$

où $\mathbf{1}_{[u_{m-1}\neq v_{n-1}]}$ vaut 1 si $u_{m-1} \neq v_{n-1}$ et 0 sinon.

E Les alignements possibles des dernières lettres pour `PECHEURS` et `ECRITURE` sont `_/E`, `S/_` ou `S/E`. Dans le premier cas, on peut aligner `PEC_HEURS` avec `_ECRITUR_` (5 désaccords), dans le deuxième `PEC_HEUR_` avec `_ECRITURE` (5 désaccord également), et dans le troisième `PEC_HEUR` avec `_ECRITUR` (4 désaccords seulement). La troisième solution est donc la meilleure, et fournit un alignement de `PECHEURS` et `ECRITURE` avec 5 désaccords au total.

Le raisonnement précédent est valable quels que soient i et j. On peut en déduire la formule récursive, valable pour $i > 0$ et $j > 0$, à laquelle on ajoute des cas de base.

Lemme 2.9 Soit u et v deux mots, et $\text{edit}(i,j)$ la distance d'édition entre $u_{[0,i[}$ et $v_{[0,j[}$. Alors $\text{edit}(0,j) = j$, $\text{edit}(i,0) = i$ et pour $i,j > 0$,

$$\text{edit}(i,j) = \min \left\{ \begin{array}{l} \text{edit}(i,j-1)+1 \\ \text{edit}(i-1,j)+1 \\ \text{edit}(i-1,j-1)+\mathbf{1}_{[u_{i-1}\neq v_{j-1}]} \end{array} \right\}.$$

Démonstration. Par définition, $\text{edit}(0,j)$ est le nombre minimal de désaccords dans un alignement du mot $u_{[0,0[}$, c'est-à-dire le mot vide, avec $v_{[0,j[}$. Il n'y a qu'une solution possible comme alignement,

consistant en j insertions de lettres de $v_{[0,j[}$. De même, $\text{edit}(i,0) = i$. Pour le cas $i,j > 0$, on va montrer les deux inégalités : $\text{edit}(i,j) \leq \min\{\ldots\}$ et $\text{edit}(i,j) \geq \min\{\ldots\}$.

Pour la première inégalité, il faut montrer que $\text{edit}(i,j)$ est inférieur ou égal à chacune des expressions du minimum. Pour aligner $u_{[0,i[}$ avec $v_{[0,j[}$, on peut aligner (optimalement) $u_{[0,i[}$ avec $v_{[0,j-1[}$ puis finir par $\_/v_{j-1}$: le nombre de désaccords sera par définition $\text{edit}(i,j-1)+1$. Puisqu'on a trouvé un alignement de $u_{[0,i[}$ avec $v_{[0,j[}$ qui a $\text{edit}(i,j-1)+1$ désaccords, le nombre minimal $\text{edit}(i,j)$ de désaccords dans un tel alignement est bien $\leq \text{edit}(i,j-1)+1$. On prouve de même que $\text{edit}(i,j) \leq \text{edit}(i-1,j)+1$ et $\text{edit}(i,j) \leq \text{edit}(i-1,j-1) + \mathbf{1}_{[u_{i-1},v_{j-1}]}$. Pour cette dernière inégalité, il faut effectuer une disjonction de cas selon que u_{i-1} et v_{j-1} sont la même lettre ou non.

Pour la seconde inégalité, on effectue le raisonnement inverse et il suffit de montrer que $\text{edit}(i,j)$ est supérieur ou égal à l'une au moins des expressions du minimum. On suppose qu'on a aligné $u_{[0,i[}$ et $v_{[0,j[}$ avec $\text{edit}(i,j)$ désaccords. Si l'alignement finit par $\_/v_{j-1}$, alors on a un alignement de $u_{[0,i[}$ avec $v_{[0,j-1[}$, qui a $\text{edit}(i,j)-1$ désaccords. Donc $\text{edit}(i,j-1) \leq \text{edit}(i,j)-1$, c'est-à-dire $\text{edit}(i,j) \geq \text{edit}(i,j-1)+1$. On obtient de même que si l'alignement finit par $u_{i-1}/\_$, alors $\text{edit}(i,j) \geq \text{edit}(i-1,j)+1$ et que s'il finit par u_{i-1}/v_{j-1}, alors $\text{edit}(i,j) \geq \text{edit}(i-1,j-1) + \mathbf{1}_{[u_{i-1} \neq v_{j-1}]}$.

Les deux inégalités permettent de conclure. □

Exercice 2.4. Calculer, à l'aide de la formule récursive, la distance d'édition entre ET et SEL.

§ **Algorithme itératif** Comme dans le cas du RENDUDEMONNAIE, transformer directement la formule récursive en un algorithme récursif ne fonctionne pas. Au contraire, on calcule la valeur de $\text{edit}(i,j)$ pour des valeurs de i et j croissantes. Pour cela, on utilise un tableau bidimensionnel E qui contient en case $E_{i,j}$ la valeur de $\text{edit}(i,j)$. On peut remplir ce tableau jusqu'à arriver à la case (m,n) qui contient le résultat souhaité. Les cases $E_{0,j}$ et $E_{i,0}$ sont remplies directement, et les autres sont remplies à l'aide de la formule récursive.

E On peut dessiner le tableau E, en indiquant les lettres des deux mots correspondant à chaque ligne et colonne. Avec PECHEURS et ECRITURE, on obtient le tableau

		E	C	R	I	T	U	R	E	
		0	1	2	3	4	5	6	7	8
P		1	1	2	3	4	5	6	7	8
E		2	1	2	3	4	5	6	7	7
C		3	2	1	2	3	**4**	5	6	7
H		4	3	2	2	3	4	5	6	7
E		5	4	3	3	3	4	5	6	6
U		6	5	4	4	4	4	4	5	6
R		7	6	5	4	5	5	5	4	5
S		8	7	6	5	5	6	6	5	5

dans lequel on lit (dernière case en bas à droite) que la distance entre ces deux mots est 5. La case **4** correspond à la distance entre ECRIT et PEC, et la valeur est obtenue à partir de la case à gauche, qui informe que la distance entre ECRI et PEC est 3 : on peut donc aligner ECRIT et PEC avec 4 désaccords, en alignant ECRI et PEC, puis T avec _.

On aboutit à l'algorithme suivant.

Algorithme 2.10 – Edition

Entrées : Deux mots u et v
Sortie : la distance d'édition entre u et v

$m \leftarrow$ taille de u ; $n \leftarrow$ taille de v
$E \leftarrow$ tableau bidimensionnel de dimensions $(m+1) \times (n+1)$

`# Cas de base`
Pour $i = 0$ à m : $E_{i,0} = i$
Pour $j = 0$ à n : $E_{0,j} = j$

`# Formule récursive`
Pour $i = 1$ à m :
  Pour $j = 1$ à n :
    $E_{i,j} \leftarrow \min(E_{i,j-1} + 1, E_{i-1,j} + 1, E_{i-1,j-1} + \mathbf{1}_{[u_{i-1} \neq v_{j-1}]})$

Renvoyer $E_{m,n}$

Théorème 2.11 L'algorithme Edition calcule la distance d'édition entre deux mots u et v en temps $O(mn)$ où m et n sont les longueurs respectives de u et v. Il utilise un espace mémoire $O(mn)$.

Démonstration. La correction de l'algorithme est directement fournie par le lemme 7. Les complexités en temps et en espace sont claires : l'algorithme se contente de remplir le tableau E, qui est de dimensions $(m+1) \times (n+1)$. Chaque case est remplie en temps $O(1)$, d'où le résultat. □

§ **Minimisation de l'espace mémoire** Si on ne cherche que la distance d'édition, on n'a pas besoin de retenir tout le tableau E. En effet, pour calculer $E_{i,j}$, on n'a besoin que des cases $E_{i,j-1}$, $E_{i-1,j}$ et $E_{i-1,j-1}$. On peut donc *oublier* toute la ligne $i-2$ à ce moment-là. Autrement dit, il suffit de ne retenir que deux lignes à chaque instant. De plus, on peut inverser u et v si v est plus long que u, pour avoir les lignes les plus courtes possibles.

Algorithme 2.12 – EditionMinMemoire

Entrées : Deux mots u et v
Sortie : la distance d'édition entre u et v

$m \leftarrow$ taille de u ; $n \leftarrow$ taille de v
Si $n > m$: échanger u et v et m et n
$P \leftarrow$ tableau de taille $(n+1)$ `# Ligne précédente`
$C \leftarrow$ tableau de taille $(n+1)$ `# Ligne courante`

`# Cas de base`
Pour $j = 0$ à n : $P_j = j$

```
    # Formule récursive
5   Pour i = 1 à m :
6     C_0 ← i # Autre cas de base
7     Pour j = 1 à n :
8       C_j ← min(C_{j-1} + 1, P_j + 1, P_{j-1} + 1_{[u_{i-1} ≠ v_{j-1}]})
9     Pour j = 0 à n : P_j ← C_j # Nouvelle ligne courante

10  Renvoyer P_n
```

Lemme 2.13 L'algorithme EditionMinMemoire calcule la distance d'édition entre deux mots u et v en temps $O(mn)$ où m et n sont les longueurs respectives de u et v. Il utilise un espace mémoire $O(\min(m,n))$.

La preuve est claire : l'algorithme est le même que Edition sauf qu'on se contente de ne retenir que les lignes nécessaires du tableau E.

§ **Reconstruction** Pour reconstruire un alignement, on part du tableau bidimensionnel E. Si on ne regarde que le coin en bas à droite, dans l'exemple, on obtient $\left[\begin{smallmatrix} 4 & 5 \\ 5 & 5 \end{smallmatrix}\right]$. La valeur de la case en bas à droite a été calculée à l'aide des trois autres cases, à la ligne 7 de l'algorithme Edition. La formule récursive pour cette case donne l'égalité $E_{m,n} = \min(E_{m,n-1}+1, E_{m-1,n}+1, E_{m-1,n-1}+\mathbf{1}_{u_{m-1}\neq v_{n-1}})$. Comme $u_{m-1} \neq v_{n-1}$ (S et E), on obtient $E_{m,n} = \min(5+1, 5+1, 4+1)$. Ainsi, on sait que le minimum était $E_{m-1,n-1} + \mathbf{1}_{u_{m-1}\neq v_{n-1}}$. Cela signifie que dans l'alignement optimal, les dernières lettres de PECHEURS et ECRITURE doivent être alignées et qu'il s'agit d'une substitution. On peut continuer le même raisonnement jusqu'à l'alignement de toutes les lettres. Cela construit une sorte de *chemin* dans le tableau E, qui part de la case $E_{m,n}$ et atteint la case $E_{0,0}$. Les directions prises par ce chemin fournissent l'alignement souhaité.

E Pour reconstruire l'alignement complet de PECHEURS et ECRITURE, on peut identifier le chemin suivant dans le tableau E (mis en évidence avec des cases en gras souligné) :

		E	C	R	I	T	U	R	E
	P	**0**	1	2	3	4	5	6	7	8
	E	**1**	1	2	3	4	5	6	7	8
	C	2	**1**	2	3	4	5	6	7	7
	H	3	2	**1**	2	3	4	5	6	7
	E	4	3	2	**2**	3	4	5	6	7
	U	5	4	3	3	**3**	**4**	5	6	6
	R	6	5	4	4	4	4	**4**	5	6
	S	7	6	5	4	5	5	5	**4**	5
		8	7	6	5	5	6	6	5	**5**

On parcourt le chemin de la case $E_{0,0}$ vers la case $E_{m,n}$. Passer d'une case $E_{i,j}$ à la case $E_{i+1,j+1}$ signifie aligner les lettres u_i et v_j. Passer d'une case $E_{i,j}$ à la case $E_{i+1,j}$ signifie aligner u_i avec un blanc, et passer d'une case $E_{i,j}$ à la case $E_{i,j+1}$ à aligner un blanc avec v_j. Dans l'exemple, on aligne donc un u_0 =P avec un blanc, puis E avec E, C avec C, R avec H, I avec E, un blanc avec T, U avec U, R avec R et enfin S avec E. On retrouve donc l'alignement PECHE_URS/_ECRITURE.

On remarque qu'il est possible qu'il y ait plusieurs possibilités à une étape (deux ou trois valeurs égales dans le calcul du minimum) : cela signifie simplement qu'il existe plusieurs alignements optimaux, et on peut choisir celui qu'on veut. D'un point de vue pratique, il est plus simple de parcourir le chemin depuis la case en bas à droite vers la case en haut à gauche, pour gérer l'indice auquel insérer des blancs.

Algorithme 2.14 – Alignement

Entrées : Deux mots u et v, tableau bidimensionnel E d'entiers
Sortie : Un alignement de u et v ayant un nombre minimal de désaccords

1. $(i,j) \leftarrow (|u|,|v|)$ `# tailles de u et v`
2. Tant que $i > 0$ et $j > 0$:
3. Si $E_{i,j} = E_{i-1,j-1}$ et $u_{i-1} = v_{j-1}$:
4. $(i,j) \leftarrow (i-1,j-1)$ `# Alignement` u_{i-1}/v_{j-1}
5. Sinon si $E_{i,j} = E_{i-1,j-1} + 1$ et $u_{i-1} \neq v_{j-1}$:
6. $(i,j) \leftarrow (i-1,j-1)$ `# Alignement` u_{i-1}/v_{j-1}
7. Sinon si $E_{i,j} = E_{i,j-1} + 1$:
8. Insérer _ en $i^{\text{ème}}$ position dans u `# Alignement _/`v_{j-1}
9. $j \leftarrow j-1$
10. Sinon : `#` $E_{i,j} = E_{i-1,j} + 1$:
11. Insérer _ en $j^{\text{ème}}$ position dans v `# Alignement` $u_{i-1}/$_
12. $i \leftarrow i-1$

`# i ou j est nul`
13. Ajouter j symboles _ en tête de u ou i symboles _ en tête de v
14. Renvoyer u et v

Lemme 2.15 Si E est le tableau rempli par Edition(u,v), Alignement(u,v,E) renvoie un alignement minimisant le nombre de désaccords, en temps $O(m+n)$ où m et n sont les tailles respectives de u et v.

Idée de la démonstration. La complexité se déduit du fait qu'à chaque passage dans la boucle, l'un des indices i, j au moins est décrémenté. Puisqu'ils valent initialement m et n, il y a au plus $m+n$ passages dans la boucle.

La correction de l'algorithme se prouve par récurrence : on démontre qu'à chaque sortie de la boucle, les fins des mots $u_{[i,m[}$ et $v_{[j,n[}$ sont optimalement alignés. On en déduit qu'à la fin de l'algorithme, u et v sont optimalement alignés. □

Pour aller plus loin

Si les deux mots ont la même longueur n, l'algorithme Edition a une complexité égale à $O(n^2)$. Peut-on calculer la distance d'édition en temps sous-quadratique ? Il existe des algorithmes qui font effectivement un peu mieux. La meilleure complexité connue est $O(n^2/\log^2 n)$. Il est en fait conjecturé qu'on ne peut pas faire *beaucoup* mieux, à savoir avoir un algorithme de complexité $O(n^{2-\epsilon})$ pour un $\epsilon > 0$. Et plus surprenant, des liens avec la question « P = NP ?

» (*cf.* 3.3) ont été exhibés récemment : si P ≠ NP dans un sens très fort[a], alors la distance d'édition ne peut se calculer en temps $O(n^{2-\epsilon})$ pour aucun $\epsilon > 0$!

a. Techniquement, l'« hypothèse forte du temps exponentiel », qui implique P ≠ NP, implique aussi la conjecture.

RÉSUMÉ

— La distance d'édition est le nombre de désaccords minimal dans un alignement de deux mots u et v, quand on peut insérer des blancs.
— On peut la calculer en temps $O(mn)$, où $m = |u|$ et $n = |v|$, et construire l'alignement en temps supplémentaire $O(m+n)$.
— L'espace mémoire de l'algorithme est $O(mn)$, mais peut-être ramené à $O(\min(m,n))$ si on ne veut que la distance, sans alignement.

2.2 Recherche textuelle

L'objet de cette partie est l'étude d'un algorithme de recherche de motif dans un texte. On dispose d'un texte (chaîne de caractères) t et d'un motif x, et l'on souhaite déterminer si le motif x apparaît dans le texte t. Plus généralement, on peut chercher à déterminer la position de x dans t si x apparaît, ou toutes les positions de x dans t. C'est cette dernière version que nous allons étudier.

Problème 2.16 – RechercheMotif

Entrées : Un texte t de longueur n, un motif x de longueur $m < n$
Sortie : Toutes les positions de x dans t

Pour formaliser les algorithmes, on introduit quelques notations. La *longueur* d'un mot w, notée $|w|$, est son nombre de lettres (caractères). On les note $w_0, \ldots, w_{|w|-1}$. On appelle *fenêtre de taille m en position i*, et on note $t_{[i,i+m[}$, le sous-mot de t constitué des lettres t_i à t_{i+m-1}. On dit qu'*un motif x apparaît dans t à la position i* si $x = t_{[i,i+m[}$, c'est-à-dire si si $x_0 = t_i$, $x_1 = t_{i+1}$, …, $x_{m-1} = t_{i+m-1}$.

E : Le motif $x = abaa$ est de longueur 4, et il apparaît dans le texte $t = acaabbabaa$ à la position 6 car $t_{[6,10[} = abaa = x$.

On commence par présenter et analyser l'algorithme naïf de recherche de motif dans un texte, puis on détaille l'algorithme de Boyer-Moore, très efficace en pratique.

2.2.1 Recherche naïve

L'algorithme naïf consiste à tester toutes les fenêtres possibles de t. Pour chaque fenêtre $t_{[i,i+m[}$, on compare x à $t_{[i,i+m[}$. Afin de préparer la présentation de l'algorithme de Boyer et Moore, on décide d'effectuer la comparaison du motif et de la fenêtre courante de droite à gauche, au lieu de la faire de manière naturelle de gauche à droite.

Algorithme 2.17 – RechercheNaïve

Entrées : Un texte t de longueur n, un motif x de longueur $m < n$
Sortie : Toutes les positions de x dans t

```
1 P ← liste vide # positions de x dans t
```

2 Pour $i = 0$ à $n-m$:
3 $\quad j \leftarrow m-1$
4 $\quad$ Tant que $j \geq 0$ et $x_j = t_{i+j}$:
5 $\quad\quad j \leftarrow j-1$
6 $\quad$ Si $j = -1$: Ajouter i à P
7 Renvoyer P

E On reprend l'exemple du motif $x = abaa$ qui apparaît en position 6 du texte $t = acaabbabaaa$. On représente ci-dessous l'exécution de RechercheNaïve sur ces entrées. Quand la fenêtre et le motif sont différents (toutes les lignes sauf $i = 6$), on représente en gras la première lettre du motif à laquelle la différence est repérée, en comparant de droite à gauche.

i	a	c	a	a	b	b	a	b	a	a	a
0	a	**b**	a	a							
1		a	b	a	**a**						
2			a	b	a	**a**					
3				a	b	**a**	a				
4					a	b	a	**a**			
5						a	b	**a**	a		
6							a	b	a	a	
7								a	**b**	a	a

Au total, 17 comparaisons sont effectuées entre deux caractères lors de l'exécution de l'algorithme.

Exercice 2.5. Programmer l'algorithme RechercheNaïve pour qu'il renvoie, en plus de la liste des positions de x dans t, le nombre de comparaisons effectuées.

Théorème 2.18 L'algorithme RechercheNaïve renvoie la liste des positions de x dans t en temps $O(mn)$.

Démonstration. On teste pour chaque fenêtre possible si $x = t_{[i,i+m[}$. L'algorithme renvoie donc bien la liste des positions de x dans t. La complexité est liée aux deux boucles imbriquées : il y a $n-m+1$ itérations de la boucle Pour, et à chaque fois au plus m itérations de la boucle Tant que, soit au plus $(n-m+1)m = O(mn)$ comparaisons et répétitions de l'étape 5 dans le pire des cas. Les autres étapes ne sont répétées que $O(n-m)$ fois. □

On remarque que le pire cas est atteint si on cherche par exemple le motif $a \cdots a$ de longueur m dans le texte $a \cdots a$ de longueur n. Si on ne souhaitait que déterminer l'existence du motif dans le texte, on atteindrait également le pire cas avec le même texte et le motif $ba \cdots a$.

Résumé

- La RechercheNaïve d'un motif de taille m dans un texte de taille n consiste à tester, pour chaque *fenêtre* du texte, si elle est égale au motif.
- Chaque comparaison d'une fenêtre avec le motif s'effectue en temps $O(m)$, donc la complexité totale est $O(mn)$.

L'algorithme de Boyer et Moore peut être vu comme une amélioration de l'algorithme naïf. L'idée est d'incrémenter plus rapidement la variable i de la boucle Pour. Pour cela, il utilise deux règles, appelées *règle du mauvais caractère* et *règle du bon suffixe*, qu'on présente l'une après l'autre.

2.2.2 La règle du mauvais caractère

Pour présenter la règle du mauvais caractère, on reprend l'exemple présenté pour RechercheNaïve.

E Lors de la comparaison pour $i = 0$, la différence entre la fenêtre et le motif est constatée lorsqu'on compare le c de la fenêtre et le b du motif. Or c n'apparaît pas dans le motif, il est donc inutile de tester le cas $i = 1$ puisqu'on aura nécessairement une différence *a minima* sur cette lettre. On peut donc directement passer à $i = 2$. Pour $i = 2$, la différence est constatée en comparant le dernier b de la fenêtre avec le dernier a du motif. À cette étape, on peut directement décaler le motif pour que ce b de la fenêtre soit *aligné* avec un b dans le motif, c'est-à-dire passer à $i = 4$. On peut continuer de même, et on obtient l'accélération suivante de l'exécution de RechercheNaïve. Dans le cas $i = 6$, on passe à $i = 7$ puisqu'on n'a constaté aucune différence avec le motif.

i	a	c	a	a	b	b	a	b	a	a	a
0	a	**b**	a	a							
2			a	b	a	**a**					
4					a	b	a	**a**			
6							a	b	a	a	
7								a	**b**	a	a

Au total, cette amélioration permet de n'effectuer que 12 comparaisons, au lieu de 17 préalablement.

Pour généraliser l'exemple et décrire la règle du mauvais caractère, on introduit la notion de *dernière occurence non finale* d'une lettre a dans un motif x : c'est la position de la dernière copie de a *la plus à droite* dans x, ou l'avant-dernière si la dernière lettre de x est a. Formellement, on note $d_x(a)$ cette position, qu'on définit par $d_x(a) = \max\{j < |x| - 1 : x_j = a\}$. Par exemple, si $x = abaa$, $d_x(a) = 2$ et $d_x(b) = 1$. Si une lettre c n'apparaît pas dans x, on pose $d_x(c) = -1$.

Le lemme suivant décrit la règle du mauvais caractère et justifie de pouvoir incrémenter i de plus de 1 dans certains cas.

> **Lemme 2.19** Soit $t_{[i,i+m[}$ la fenêtre courante, et $j > 0$ un indice tel que $x_j \neq t_{i+j}$ et $x_k = t_{i+k}$ pour $k > j$. Soit $d = d_x(t_{i+j})$ la position de la dernière occurrence non finale de t_{i+j} dans x. Alors si $j > d$, x est différent de la fenêtre $t_{[i_0,i_0+m[}$ pour $i < i_0 < i + j - d$.

Démonstration. On peut suivre le raisonnement sur la figure 4. On considère une position i_0 entre $i + 1$ et $i + j - d - 1$. On veut montrer que $x \neq t_{[i_0,i_0+m[}$. La lettre t_{i+j} appartient à cette fenêtre et elle est alignée avec la lettre x_{i-i_0+j} du motif (en mauve sur la figure). Or si $i < i_0 < i + j - d$, x_{i-i_0+j} est *à droite* de x_d (case verte). Mais par définition de d, x_d est la case *la plus à droite* qui est égale à t_{i+j}, donc la lettre x_{i-i_0+j} (en mauve) est différente de t_{i+j}. Donc le motif est différent de la fenêtre. □

Il se peut que j soit inférieur à d. Dans ce cas, la règle du mauvais caractère ne sert à rien et on incrémente i de 1 comme dans RechercheNaïve. On obtient une version simplifiée de l'algorithme de Boyer et Moore, qu'on note BoyerMoore-MC pour « Mauvais Caractère ».

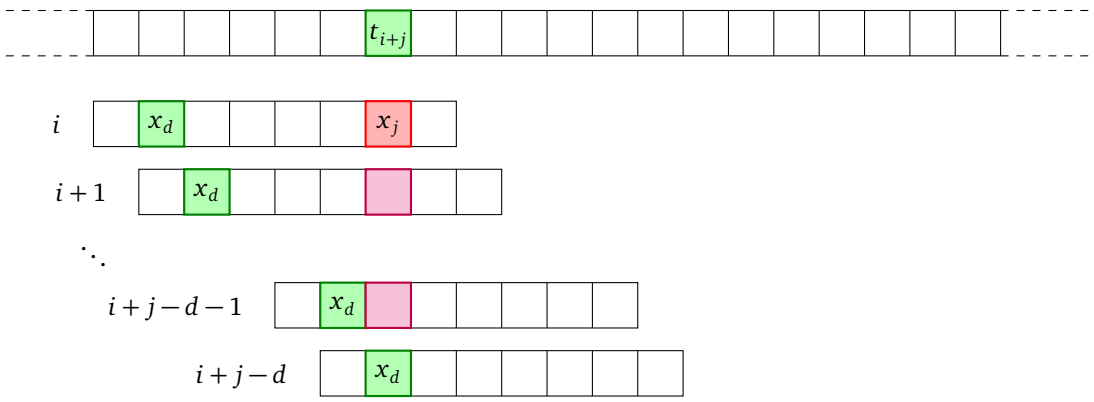


FIGURE 4 – À la position i, le motif est différent de la fenêtre car $x_j \neq t_{i+j}$. L'occurrence non finale la plus à droite de t_{i+j} dans x est x_d. Si on décale le motif en position $i_0 < i+j-d$, le motif est encore différent car $t_{i+j} \neq x_{i-i_0+j}$ (case mauve). Mais en position $i+j-d$, x_d est aligné avec t_{i+j}.

Algorithme 2.20 – BOYERMOORE-MC

Entrées : Un texte t de longueur n, un motif x de longueur $m < n$
Entrée supplémentaire : Un tableau associatif d qui contient la position de dernière occurence non finale dans x pour chaque lettre (qu'elle apparaisse ou non dans x)
Sortie : Toutes les positions de x dans t

```
1  P ← liste vide # positions de x dans t
2  i ← 0
3  Tant que i ≤ n − m :
4      j ← m − 1
5      Tant que j ≥ 0 et x_j = t_{i+j} :
6          j ← j − 1
7      Si j = −1 :
8          Ajouter i à P
9      i ← i + max(1, j − d[t_{i+j}])
10 Renvoyer P
```

Théorème 2.21 L'algorithme BOYERMOORE-MC renvoie la liste des positions de x dans t en temps $O(mn)$.

Démonstration. La correction de l'algorithme découle directement du lemme 19. Son analyse de complexité est similaire à celle de la RECHERCHENAÏVE : les deux boucles imbriquées ont un coût dans le pire cas de $O(mn)$. □

Comme indiqué préalablement, on peut atteindre la même complexité dans le pire des cas que la RECHERCHENAÏVE avec cet algorithme. Un exemple d'entrée qui atteint cette complexité est un motif

$ba\cdots a$ que l'on recherche dans un texte $a\cdots a$: à chaque passage dans la boucle externe, il faut comparer le motif et la fenêtre en entier (puisqu'on commence par la fin), et le décalage n'est que de 1 puisque la position de dernière occurrence non finale de a dans le motif est $m-2$. On pourrait exhiber un exemple similaire si on comparait les motifs de gauche à droite.

Il reste à savoir calculer efficacement l'*entrée supplémentaire*, c'est-à-dire le tableau d des dernières occurences non finales. Cela se fait aisément en temps $O(n)$.

Exercice 2.6.

1. Écrire un algorithme qui calcule un tableau associatif d tel que pour toute lettre a présente dans le motif, $d[a]$ est la position de la dernière occurrence non finale de a dans le motif.
2. L'algorithme simplifié de Boyer et Moore nécessite d'avoir également $d[b]=-1$ pour toutes les lettres b qui n'apparaissent pas dans le motif. On pourrait donc étendre le tableau d pour qu'il contienne toutes les lettres possibles. Quel est l'inconvénient de cette solution ?
3. Quel type de structure de données peut-on utiliser en Python pour remédier à cette difficulté ?
4. Écrire un programme qui implante l'algorithme BoyerMoore-MC en comptant les comparaisons effectuées.

Résumé

— La *règle du mauvais caractère* consiste à calculer un décalage de fenêtre en cherchant dans le motif, une lettre correspondant à la lettre de la fenêtre du texte grâce à laquelle la différence avec le motif a été constatée.
— Pour identifier la lettre voulue, on utilise un tableau qui peut être précalculé en temps $O(m)$ où m est la taille du motif. Pour cela, il est crucial que le tableau ne dépende que du motif.

2.2.3 Règle du bon suffixe

Avant d'introduire la deuxième règle, on remarque qu'on pourrait définir une ou plusieurs variantes de la règle du mauvais caractère. En effet, cette règle s'appuie sur le caractère t_{i+j}. Mais on pourrait de manière équivalente raisonner sur la lettre t_{i+m-1} (la dernière lettre de la fenêtre) : on cherche la dernière occurrence non finale x_d de t_{i+m-1} dans x, et on calcule le décalage pour que x_d soit aligné avec t_{i+m-1}. Cette variante est appelée *algorithme de Horspool*. Et on pourrait également raisonner sur *toutes* les lettres qu'on a comparé (t_{i+j} à t_{i+m-1}). Chacune fournirait une valeur de décalage et il suffirait d'effectuer le plus grand de ces décalages. L'inconvénient de cette technique est que ce calcul du maximum coûterait à chaque étape $O(m)$, et qu'on n'aurait ainsi aucune chance d'avoir un algorithme plus rapide que $O(mn)$.

La *règle du bon suffixe* cherche à tirer partie de toutes ces lettres plus efficacement. Si on a comparé le motif x à la fenêtre $t_{[i,i+m[}$ et qu'on a constaté une différence $x_j \neq t_{i+j}$, on appelle *bon suffixe* le mot $t_{[i+j+1,i+m[}$, qui est par définition égal à $x_{[j+1,m[}$.

E On cherche le motif $x=abaaaa$ dans le texte $t=abbcaacaaaabaaaa$ (x apparaît à la fin de t, en position 10). Pour $i=0$, on compare le motif avec la fenêtre $abbcaa$. Le *bon suffixe* de la comparaison est $t_{[4,6[}=aa$. Pour que la prochaine ait une chance d'aboutir, il faut *a minima* que ce bon suffixe de la fenêtre se retrouve aligné avec une autre copie de aa dans le motif. Ces copies se trouvent aux positions 2 et 3. On peut être tenté de décaler d'un cran pour aligner le bon suffixe avec la copie $x_{[3,5[}$ de aa la plus à droite dans le motif, de manière un peu similaire à la règle du mauvais caractère. Mais cette copie de aa est précédée d'un a ($x_2=a$), tout comme le suffixe $x_{[4,6[}=aa$ du motif : si on effectue la comparaison, on est sûr de tomber encore sur une différence, car $x_2=a$ sera aligné avec $t_3=c$. On décale donc jusqu'à aligner le bon suffixe

$t_{[4,6[}$ avec une copie de aa qui soit précédée par une autre lettre que celle qui précède le suffixe $x_{[4,6[}$ du motif. Ici, on peut donc décaler de deux crans, et aligner le motif avec la fenêtre en position 2.
On compare donc maintenant x à la fenêtre $bcaaca$. Le bon suffixe est a, et il est précédé d'un a dans x. Sa copie la plus à droite dans x qui n'est pas précédée d'un a est en position 2. On peut donc décaler le motif pour aligner x_2 avec le dernier a de la fenêtre, c'est-à-dire passer directement à $i = 5$.
On effectue alors la comparaison de x avec $acaaaa$. Cette fois-ci, le bon suffixe est $aaaa$. Il n'apparaît pas ailleurs dans le motif. Pour calculer le décalage, on utilise une règle un peu différente : on cherche le *préfixe* de x le plus long possible qui soit aussi suffixe du bon suffixe. Dans notre cas, c'est a puisque d'une part, x commence par a et le bon suffixe finit par a, et d'autre part x commence par ab alors que le bon suffixe ne finit pas par ab. On aligne ce préfixe de x avec le suffixe correspondant du bon suffixe, c'est-à-dire ici qu'on passe à $i = 10$ directement.

i	a	b	b	c	a	a	c	a	a	a	a	b	a	a	a	a
0	a	b	a	**a**	a	a										
2			a	b	a	a	**a**	a								
5						a	**b**	a	a	a	a					
10											a	b	a	a	a	a

La *règle du bon suffixe* est la formalisation des deux règles utilisées dans l'exemple : on recherche une copie du bon suffixe, précédée d'une lettre différente, la plus à droite possible dans le motif ; si on n'en trouve pas, on considère le plus long préfixe du motif qui soit aussi suffixe du bon suffixe. On note de manière cruciale que, comme pour la règle du mauvais caractère, cela ne dépend que du motif et peut donc être calculé avant de parcourir le texte.

Pour exprimer formellement la règle, on note $p_x(j)$ la longueur du plus long préfixe de x qui soit aussi suffixe de $x_{[j,m[}$. On impose que le préfixe soit différent de x entier, et on pose donc $p_x(0) = p_x(1)$. De plus, on note $s_x(j)$ la position la plus à droite d'une copie du suffixe $x_{[j,m[}$ qui ne soit pas précédée par la lettre x_{j-1} : $s_x(j) = \max\{k < j : x_{[k,k+m-j[} = x_{[j,m[}$ et $x_{k-1} \neq x_{j-1}\}$ où l'on prend pour convention que x_{-1} est différent de toute lettre. Si un tel k n'existe pas, on pose $s_x(j) = -1$. On pose $s_x(m) = m - 1$ par convention.

E Avec le motif précédent $abaaaa$, $p_x(j) = 1$ pour tout $0 < j < 6$. On a également $s_x(3) = s_x(4) = s_x(5) = 2$, et $s_x(j) = -1$ pour $j < 3$.

Lemme 2.22 Soit $t_{[i,i+m[}$ la fenêtre courante, et $j \geq 0$ un indice tel que $x_{[j+1,m[} = t_{[i+j+1,i+m[}$ et $x_j \neq t_{i+j}$. Soit $s = s_x(j+1)$ et $p = p_x(j+1)$. Alors $x \neq t_{[i_0,i_0+m[}$ pour $i < i_0 < i + j + 1 - s$ si $s \geq 0$, et $x \neq t_{[i_0,i_0+m[}$ pour $i < i_0 < i + m - p$ sinon.

Démonstration. Le raisonnement peut se suivre sur les figures 5 et 6. On se place dans le premier cas, $s \geq 0$ (figure 5). Si on décale le motif en $i_0 < i + j + 1 - s$, il ne peut pas être égal à la fenêtre. En effet, les cases $x_{[i+j-i_0,i+m-i_0[}$ se retrouvent alignées avec $t_{[i+j,i+m[}$. Ces deux sous-fenêtres ne peuvent pas être égales, sinon on aurait une copie du bon suffixe, précédée d'une lettre différente de x_j (car on aurait $x_{i+j-i_0} = t_{i+j} \neq x_j$), à une position $i + j + 1 - i_0 > s$.

Si maintenant $s = -1$ (figure 6), on sait avec le même raisonnement que précédemment qu'on ne peut pas décaler le motif en position $i_0 < i + j$ (car le bon suffixe ne se retrouve pas dans x). De plus,

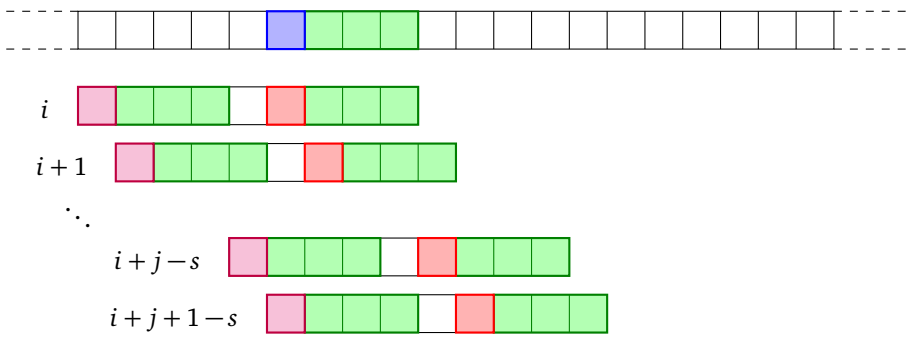


FIGURE 5 – À la position i, on constate un bon suffixe de longueur 3. La copie du suffixe la plus à droite dans x qui est précédée d'une case différente est en position $s = 1$. Si on décale le motif en position $i_0 < i + j + 1 - s$, le motif est encore différent de la fenêtre. En position $i + j + 1 - s$, les trois cases du bon suffixe sont alignées avec des cases correspondantes dans x.

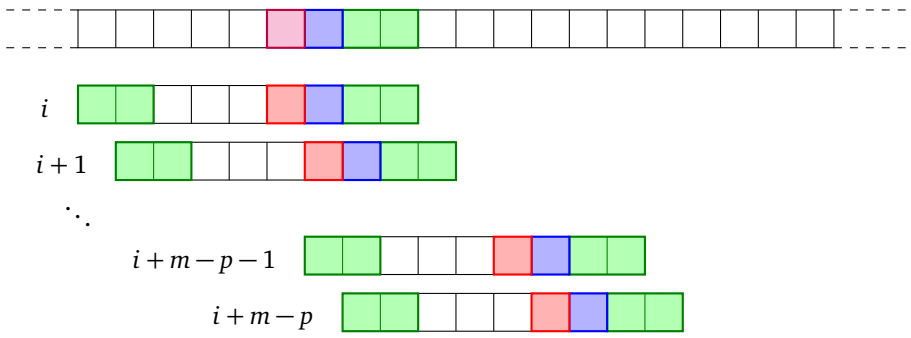


FIGURE 6 – À la position i, on constate un bon suffixe de longueur 3, qui n'apparaît nulle part ailleurs dans le motif. Le plus long préfixe de x qui est également suffixe du bon suffixe est de taille 2 (en vert). Si on décale le motif en position $i_0 < i + m - p$, le motif est encore différent de la fenêtre. En position $i + m - p$, les deux dernière cases du bon suffixe sont alignées avec le préfixe correspondant dans x.

si x apparaît dans t en position i_0, cela signifie en particulier que $t_{[i_0,i+m[} = x_{[0,i+m-i_0[}$. Si $i_0 \geq i+j$, on sait que $t_{[i_0,i+m[} = x_{[i_0-i,m[}$ car $t_{[i+j,i+m[} = x_{[j,m[}$. On a donc $x_{[0,i+m-i_0[} = x_{[i_0-i,m[}$. La première position i_0 à laquelle x peut potentiellement apparaître fournit donc un préfixe de x de longueur $i+m-i_0$ qui est également suffixe du bon suffixe. Autrement dit, la longueur maximale d'un tel préfixe vérifie $p \geq i+m-i_0$, c'est-à-dire $i_0 \geq i+m-p$. □

On en déduit une nouvelle version simplifiée de l'algorithme de Boyer et Moore.

Algorithme 2.23 – BoyerMoore-BS

Entrées : Un texte t de longueur n, un motif x de longueur $m < n$
Entrées supplémentaires : Deux tableaux s et p tels que $s[j] = s_x(j)$ et $p[j] = p_x(j)$
Sortie : Toutes les positions de x dans t

```
1  P ← liste vide # positions de x dans t
2  i ← 0
3  Tant que i ≤ n − m :
4      j ← m − 1
5      Tant que j ≥ 0 et x_j = t_{i+j} :
6          j ← j − 1
7      Si j = −1 :
8          Ajouter i à P
9          i ← i + m − p[1]
10     Sinon :
11         Si s[j + 1] ≥ 0 : i ← i + j + 1 − s[j + 1]
12         Sinon : i ← i + m − p[j]
13 Renvoyer P
```

Théorème 2.24 L'algorithme BoyerMoore-BS renvoie la liste des positions de x dans t en temps $O(mn)$.

On omet la preuve, qui est identique à celle pour l'algorithme BoyerMoore-MC. On remarque qu'encore une fois, on n'a pas obtenu un algorithme meilleur que RechercheNaïve dans le pire des cas.

Il reste un point délicat : comment peut-on calculer efficacement les tableaux s et p ? Il est assez facile, pour chacun des deux tableaux, de les calculer en temps $O(m^2)$. On peut en fait les calculer en temps linéaire $O(m)$, mais on n'aborde pas ce point ici.

Exercice 2.7.

1. Écrire un algorithme qui étant donné j, teste si $x_{[0,m-j[} = x_{[j,m[}$.
2. En déduire un algorithme de complexité $O(m^2)$ qui prend en entrée le motif x et renvoie le tableau p.
3. Écrire un algorithme qui étant donné j, calcule la longueur ℓ du plus long suffixe de x tel que $x_{[m-\ell,m[} = x_{[j-\ell,j[}$.
4. En déduire un algorithme de complexité $O(m^2)$ qui prend en entrée le motif x et renvoie le tableau s.
5. Écrire un programme qui implante l'algorithme BoyerMoore-BS en comptant les comparaisons effectuées.

Résumé

- La *règle du bon suffixe* calcule un décalage en alignant le suffixe de la fenêtre sur une copie de ce suffixe dans le motif, et si c'est impossible en alignant le suffixe sur le plus long préfixe possible du motif.
- Cette règle peut être appliquée à l'aide de deux tableaux, précalculés en fonction du motif.
- Le précalcul s'effectue facilement en temps $O(m^2)$ où m est la longueur du motif, et plus difficilement en temps $O(m)$.

2.2.4 Algorithme de Boyer et Moore

L'algorithme de Boyer et Moore est obtenu en combinant les deux règles présentées précédemment. À chaque étape, on effectue un décalage en appliquant *la plus favorable* des deux règles.

Algorithme 2.25 – BoyerMoore

Entrées : Un texte t de longueur n, un motif x de longueur $m < n$
Entrées supplémentaires : Un tableau associatif d tel que $d[a] = d_x(a)$, et deux tableaux s et p tels que $s[j] = s_x(j)$ et $p[j] = p_x(j)$
Sortie : Toutes les positions de x dans t

```
1  P ← liste vide # positions de x dans t
2  i ← 0
3  Tant que i ≤ n − m :
4      j ← m − 1
5      Tant que j ≥ 0 et x_j = t_{i+j} :
6          j ← j − 1
7      Si j = −1 :
8          Ajouter i à P
9          i ← i + m − p[1]
10     Sinon :
11         Si s[j + 1] ≥ 0 : i ← i + max(j + 1 − s[j + 1], j − d[t_{i+j}])
12         Sinon : i ← i + max(m − p[j], j − d[t_{i+j}])
13 Renvoyer P
```

Théorème 2.26 L'algorithme BoyerMoore renvoie la liste de toutes les positions de x dans t en temps $O(mn)$.

Tout ça pour ça ! Effectivement, l'algorithme de Boyer et Moore a dans le pire cas encore la même complexité que la recherche naïve. Mais cette complexité est un peu trompeuse. Non seulement il est très rapide en pratique, mais surtout on peut montrer que dans un très grand nombre de cas, sa complexité est $O(m+n)$. Il existe même une modification de l'algorithme qui dans tous les cas atteint cette complexité. Ces considérations sont bien plus avancées.

Exercice 2.8. Écrire un programme qui implante l'algorithme BoyerMoore en comptant les comparaisons effectuées.

Résumé

— L'algorithme de Boyer et Moore améliore la recherche naïve, en appliquant les règles du mauvais caractère et du bon suffixe, pour avancer aussi vite que possible dans le texte.
— Sa complexité dans le pire des cas reste celle de la recherche naïve.
— Cependant, il est efficace en pratique et on peut démontrer que dans de nombreux cas *non pathologiques*, sa complexité est $O(m+n)$ où m est la taille du motif et n celle du texte.
— Il existe une variante, appelée *règle de Galil*, qui atteint cette complexité dans tous les cas.

Pour aller plus loin

Il existe d'autres algorithmes de recherche de motif dans un texte qui ont la même complexité $O(m+n)$. Parmi les plus connus, on peut citer celui de Knuth, Morris et Pratt basé sur la notion d'automate fini, ou celui de Karp et Rabin utilisant une technique d'*empreinte probabiliste*. Ces algorithmes sont utilisés dans vos logiciels préférés pour faire une recherche de texte. Il peuvent être étendus pour permettre la recherche d'*expressions régulières* (motifs avec des lettres non fixées par exemple), et sont extrêmement utiles en bio-informatique, pour identifier des gênes dans une séquence ADN par exemple.

3 Calculabilité et complexité

Les théories de la calculabilité et de la complexité sont deux pans de la théorie du calcul : qu'est-ce que veut dire *calculer* ? que peut-on calculer ? que peut-on calculer efficacement ? La calculabilité apporte des réponses aux deux premières questions. La complexité s'intéresse à la troisième et peut être vue comme une version quantitative de la calculabilité.

En algorithmique, l'objet d'étude principal est l'*algorithme*. On étudie par exemple pour un algorithme donné sa *complexité* (en temps, ou en espace, etc.). Parfois, plusieurs algorithmes résolvent un même *problème* et on cherche à les comparer. En calculabilité et complexité, le problème devient l'objet central.

La première partie introduit le vocabulaire et les distinctions entre *algorithmes*, *problèmes* et *fonctions*. La deuxième partie est une introduction à la calculabilité et la troisième à la complexité. Enfin, la quatrième partie, optionnelle, présente des notions plus avancées et quelques liens bibliographiques pour aller plus loin.

3.1 Algorithmes, problèmes, fonctions

3.1.1 Définitions

§ **Algorithme** La théorie de la calculabilité permet de donner une définition très précise de ce qu'est un algorithme. Plus précisément, la calculabilité *se base sur une définition formelle, ou modèle, d'algorithme*. Son succès est grandement lié au fait que toutes les définitions proposées se sont avérées strictement équivalentes.

Dans ce texte, le modèle utilisé est celui des *programmes Python* : la formalisation d'un algorithme qu'on utilisera principalement consistera en son écriture comme *fonction Python* (définie avec le mot-clef `def`). Dans toute la suite, on parlera de *programme* ou d'*algorithme* car on réserve le mot *fonction* pour un autre usage. Par exemple, l'algorithme d'Euclide pour calculer le PGCD de deux entiers peut être défini par le programme suivant.

```
def euclide(a,b):
    if a<b: a,b=b,a
    while b: a,b=b,a%b
    return a
```

R Pour faire de la calculabilité, on est tiraillé entre deux envies pour le modèle qu'on choisit : d'un côté, on le souhaite expressif et *facile à utiliser* ; d'un autre côté, on le souhaite aussi rudimentaire que possible pour pouvoir effectuer des preuves. On a choisi Python, c'est-à-dire une modélisation très expressive et absolument pas rudimentaire, afin de pouvoir effectuer les preuves en restant à un niveau très intuitif.

§ **Fonction** *Un algorithme calcule une fonction*, qui va de l'ensemble des entrées possibles vers l'ensemble des sorties possibles. L'algorithme précédent calcule la fonction pgcd : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ telle que pgcd(a, b) est le PGCD de a et b. Il y a une différence importante entre l'algorithme et la fonction qui est calculée car *plusieurs algorithmes peuvent calculer la même fonction*. Par exemple, la fonction pgcd peut être calculée par d'autres algorithmes, comme l'algorithme inefficace suivant.

```
def pgcd_inefficace(a, b):
    g = a
    while a % g != 0 or b % g != 0:
        g -= 1
    return g
```

§ **Fonction partielle** Les fonctions calculées par des algorithmes sont des *fonctions partielles* : si un programme *boucle* sur une entrée x, la fonction n'est pas définie en x. On note $f(x) = \perp$ pour dire que f n'est pas définie en x. Par exemple, la fonction définie par l'algorithme suivant n'est définie que pour les entiers pairs positifs ou nuls car l'algorithme *boucle* sur une entrée impaire ou strictement négative. On dit qu'un programme *termine* s'il ne boucle pas, c'est-à-dire qu'il renvoie bien un résultat.

```
def partiel(x):
    c = 0
    while x != 0:
        x -= 2
        c += 1
    return c
```

R En mathématiques, on ne parle en général pas de fonction partielle. On définit le *domaine de définition* de la fonction comme l'ensemble des valeurs sur lesquelles la fonction est définie. De cette manière, une fonction est toujours *totale* quand on la restreint à son domaine de définition. En calculabilité, l'utilisation de fonctions partielles est nécessaire pour étudier les programmes qui bouclent. Au lieu de voir la fonction précédente comme une fonction définie sur $2\mathbb{N}$, la voir comme une fonction définie sur $\mathbb{Z}$ où $f(x) = \perp$ pour tout $x \notin 2\mathbb{N}$.

Une fonction *totale* est une fonction définie sur toute entrée. En calculabilité, une fonction est *a priori* partielle, et le fait d'être *totale* est une caractéristique de certaines fonctions partielles.

§ **Problème de décision** Informatiquement, l'appellation *problème* est un synonyme de fonction partielle : dire qu'un algorithme résout le problème du PGCD signifie simplement que la fonction calculée par l'algorithme est la fonction pgcd. Parmi ces problèmes, *un problème de décision est une fonction partielle à valeur dans* $\{0,1\}$. Autrement dit, c'est un problème dont la sortie est soit 0 soit 1, ou de manière équivalente OUI/NON ou VRAI/FAUX, etc. En Python, on utilisera les valeurs `True` et `False` comme valeurs de retour pour des algorithmes calculant des problèmes de décision. Des exemples de problèmes de décision peuvent être : l'entier n est-il premier ? la chaîne de caractères s est-elle un palindrome [13] ? le graphe G est-il connexe ?

§ **Fonction et problèmes calculables** Enfin, la dernière notion importante est celle de *fonction (partielle) calculable* ou *problème calculable* : c'est une fonction pour laquelle il existe un algorithme. Puisque les fonctions sont partielles, dire qu'un algorithme calcule une fonction signifie que si la fonction f est définie sur une entrée x, alors l'algorithme doit renvoyer la valeur $f(x)$, et que si f n'est pas définie, l'algorithme doit boucler sur l'entrée x. Par exemple, l'algorithme `partiel` ci-dessus calcule la fonction partielle $f : \mathbb{N} \to \mathbb{N}$ définie par

$$f(x) = \begin{cases} \frac{x}{2} & \text{si } x \text{ est un entier pair positif ou nul, et} \\ \bot & \text{sinon.} \end{cases}$$

RÉSUMÉ

- Un *algorithme* est (dans ce texte) un programme Python.
- Une *fonction partielle* est une fonction qui peut être indéfinie sur certaines entrées.
- Un *problème de décision* est une fonction partielle à valeur dans $\{0,1\}$.
- Une fonction partielle est *calculable* s'il existe un algorithme pour la calculer.
- Un algorithme calcule une fonction, et plusieurs algorithmes peuvent calculer la même fonction.

3.1.2 Algorithmes comme données

Un point important en calculabilité est la possibilité de manipuler un algorithme comme une donnée comme une autre. Chaque programme Python est défini par sa chaîne de caractères. Par exemple, le programme `euclide` précédent est défini par la chaîne de caractères [14]

```
"def euclide(a,b):\n\tif a < b: a,b=b,a\n\twhile b: a,b=b,a%b\n\treturn a".
```

Cela signifie en particulier qu'on peut effectuer des calculs sur un algorithme puisque c'est une chaîne de caractères comme une autre. Bien entendu, on peut effectuer sur un algorithme des calculs purement syntaxiques, tels que renommer une variable `x` en `y`. Mais l'importance de l'utilisation des algorithmes comme données est lié au fait qu'on peut faire des calculs bien plus intéressants.

R Il peut y avoir une subtilité entre nom d'un algorithme (`euclide` ci-dessus) et le nom qu'on donne à la chaîne de caractères qui le décrit (`"def euclide(a,b): ..."`). Dans la suite, on notera <algo> la chaîne de caractères qui décrit l'algorithme `algo`. Par exemple <euclide> est la chaîne `"def euclide(a,b): ..."`.

13. Un *palindrome* est une chaîne qui peut se lire dans les deux sens, comme `laval`, `radar` ou `001100`.
14. Les symboles `\n` et `\t` représentent le passage à la ligne et la tabulation, respectivement.

§ **Programme universel** Il est possible d'*interpréter* un algorithme sur une entrée donnée, c'est-à-dire de simuler son comportement. C'est ce que fait le programme suivant qui prend en entrée une chaîne de caractères représentant un programme, ainsi que des arguments, et évalue le programme sur les arguments [15].

```
def universel(algo, *args):
    exec(algo)
    ligne1 = algo.split('\n')[0]
    nom = ligne1.split('(')[0][4:]
    return eval(f"{nom}{args}")
```

On peut l'utiliser de la manière suivante.

```
pg='def euclide(a,b):\n\tif a<b:a,b=b,a\n\twhile b:a,b=b,a%b\n\treturn a'
universel(pg,35,49)
7
```

Utiliser `exec` et `eval` peut sembler être de la triche... En effet, on cache toute la complexité de ce programme `universel` dans l'utilisation de ces deux fonctions. Le point important (qu'on ne prouvera pas) est qu'il est *possible* d'écrire ces fonctions `exec` et `eval` directement en Python. Autrement dit, il est possible d'écrire le programme `universel` sans faire appel à ces deux fonctions. Cela revient à *écrire un interpréteur Python en Python*. Le fait que ce soit possible en Python, ou pour n'importe quel langage de programmation, est l'un des résultats centraux de la calculabilité. Pour le démontrer, ce qu'on ne fera pas, il vaudrait mieux utiliser une formalisation plus rudimentaire d'algorithme. On reviendra sur cette question plus loin.

Une conséquence de ce programme `universel` est qu'étant donné le code d'un programme, c'est-à-dire la chaîne de caractères qui le représente, on sait exécuter le programme sur une entrée. En particulier, un programme `pgm` peut faire appelle à un autre programme `autre` si on fournit la chaîne de caractères `<autre>` à `pgm`.

En pratique

Les programmes qui manipulent (le code) d'autres programmes sont courants : c'est par exemple le cas des compilateurs. La notion de programme universel est le pendant théorique des interpréteurs en informatique. Le point délicat à accepter est qu'un interpréteur pour un langage de programmation donné peut être écrit dans le langage lui-même ! La notion de compilateur est légèrement différente mais admet aussi son pendant théorique, voir 3.4.5.

Résumé

- Un algorithme est une donnée comme une autre, manipulable par des algorithmes.
- Un *algorithme universel* permet d'exécuter tout algorithme sur n'importe quelle(s) entrée(s).

3.2 Calculabilité

La calculabilité, apparue dans les années 1930 avec les travaux fondateurs de Turing, Church, Kleene, et d'autres, s'intéresse à savoir quels problèmes de décision sont *calculables*, c'est-à-dire

15. Comprendre le détail de son fonctionnement n'est pas nécessaire.

lesquels peuvent être calculés par un algorithme. Pour cela, cette théorie propose une formalisation de la notion d'algorithme. L'année 1936 a été un point de bascule car différentes formalisations d'algorithmes ont été proposées, qui se sont avérées être parfaitement équivalentes bien qu'elles ne se ressemblent absolument pas. Dans toute cette partie, le terme *fonction* signifie *fonction partielle* quand on ne le précise pas.

3.2.1 Toutes les fonctions ne sont pas calculables

Un des premiers résultats de calculabilité est l'existence de fonctions explicites non calculables. Le fait qu'il existe des fonctions non calculables est lié à la notion mathématique de *dénombrabilité*. Un algorithme, défini comme un programme Python, est décrit par une chaîne de caractères finie. Cela implique que l'ensemble des algorithmes est un ensemble *dénombrable*. À l'inverse, on peut montrer que l'ensemble des problèmes de décision est un ensemble *indénombrable*, ce qui implique que certains problèmes de décision n'admettent pas d'algorithme : il n'y a tout simplement pas assez d'algorithmes ! *Voir la partie 3.4.1 pour les définitions et des détails sur cet argument.*

On peut même dire un peu plus : la quasi-totalité des problèmes de décision est incalculable ! On cherche maintenant à décrire explicitement un tel problème, ce qui est souvent résumé par l'objectif paradoxal : *trouver du foin dans une botte de foin*.

La première fonction explicite non calculable a été décrite par Turing en 1936. Il s'agit du *problème de l'arrêt* : étant donné un algorithme A et une entrée m, il s'agit de déterminer si A s'arrête ou non sur l'entrée m (*i.e* A ne boucle pas sur l'entrée m). C'est un problème qu'on se pose souvent en algorithmique, quand on veut démontrer qu'un algorithme termine. Cependant, on peut construire des exemples très simples pour lesquels démontrer la terminaison est extrêmement difficile. Par exemple, personne ne sait dire si le programme suivant termine, car sa terminaison est équivalente à la conjecture de Syracuse, problème ouvert en mathématiques.

```
def syracuse(n):
    while n != 1:
        if n % 2 == 0: n = n // 2
        else:          n = 3*n + 1
    return True
```

L'approche évidente pour résoudre le problème de l'arrêt consiste à utiliser le programme `universel` pour simuler A sur l'entrée m. Si A s'arrête sur l'entrée m, on peut renvoyer `True`. Mais si ce n'est pas le cas, `universel` ne répondra jamais. À quel moment décide-t-on que si `universel` n'a *pas encore* répondu, c'est qu'il ne répondra jamais ? On n'a aucun moyen de le faire, et c'est ce que prouve le théorème de Turing.

Théorème 3.1 Le problème de l'arrêt est incalculable.

Démonstration. On suppose que le problème est calculable, c'est-à-dire qu'il existe un programme `arret` qui prend en entrée le code d'un algorithme A et une entrée m, et répond `True` si A termine sur l'entrée m, et `False` sinon.

```
def arret(algo, m):
    "Renvoie True si algo termine sur l'entrée m, False sinon"
```

```
    ... # comment faire ?
```

Bien sûr, toute la difficulté est de remplir les « ... ». Mais supposons un instant qu'on sache le faire. On peut alors construire l'algorithme paradoxal suivant.

```
def diagonal(algo):
    if not universel(<arret>, algo, algo):  # not arret(algo, algo)
        return True
    while True: # on boucle
        pass
```

L'algorithme `diagonal` est parfaitement défini. On peut donc l'appeler sur n'importe quelle entrée, en particulier sur son propre code `<diagonal>`. Que fait l'appel `diagonal(<diagonal>)` ?

— Si cet appel termine, alors `arret(<diagonal>,<diagonal>)` répond `True`. La condition du test n'est donc pas vérifiée, et on rentre dans la boucle infinie.
— Si cet appel ne termine pas, alors `arret(<diagonal>,<diagonal>)` répond `False` et on répond `True`.

On arrive donc à une contradiction. Si `diagonal(<diagonal>)` termine alors il rentre dans une boucle infinie, et s'il ne termine pas il répond immédiatement `True`. C'est impossible, donc l'hypothèse de départ, l'existence de l'algorithme `arret`, est fausse. □

En pratique

La non-calculabilité du problème de l'arrêt signifie qu'on ne peut pas avoir de programme qui détecte toutes nos erreurs de programmation, en particulier des boucles infinies non voulues. Le *théorème de Rice* va même plus loin en démontrant qu'en réalité, un programme comme un compilateur ne peut analyser du code que de manière très superficielle, et non pas *comprendre* ce que fait ce code. Voir 3.4.3.

Résumé

— L'ensemble des algorithmes, donc des fonctions calculables, est dénombrable.
— L'ensemble des fonctions est indénombrable, donc il existe des fonctions incalculables.
— Le problème de l'arrêt est un exemple de fonction incalculable.

3.2.2 Formalisation : la machine de Turing

Les discussions précédentes ont gardé la notion d'algorithme à un niveau intuitif, en recourant au langage Python. En réalité, l'ensemble des programmes Python peut servir de définition tout à fait formelle de la notion d'algorithme. Son inconvénient est sa complexité. Avoir des formalisations plus simples permet de prouver plus facilement des résultats tels que l'existence d'un programme universel. On présente la formalisation la plus connue de la notion d'algorithme : la machine de Turing (1936). *On admet que cette formalisation donne lieu aux mêmes fonctions calculables, exactement, que les programmes Python*. Un sens de cet équivalence est démontré en exercice à la fin de cette partie.

§ **Description** La machine de Turing est une machine théorique capable d'effectuer des calculs. Elle est constituée des éléments suivants :

- un *ruban* infini, constitué d'une infinité de cases pouvant contenir des symboles 0, 1 ou □ (case vide) ;
- un ensemble fini d'*états* ;
- une tête de lecture/écriture, qui à tout instant se trouve au dessus d'une case du ruban et peut se déplacer de cases en cases vers la gauche ou la droite ;
- une *table de transition* qui définit le comportement de la machine : en fonction de l'état courant et du symbole lu par la tête, la machine écrit un nouveau symbole sur le ruban, déplace sa tête de lecture (d'une case à gauche ou à droite) et passe dans un nouvel état.

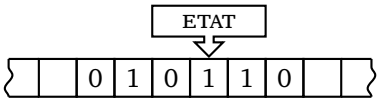


FIGURE 7 – Représentation d'un ruban de machine de Turing : six cases sont non vides, la tête de lecture est au dessus d'un 1, et la machine est dans l'état ETAT.

§ **Calcul sur une machine de Turing** Un calcul se déroule de la manière suivante. On débute dans l'*état initial* INIT avec l'entrée (finie) écrite sur le ruban (toutes les cases sauf un nombre fini sont vides), et la tête se trouve au dessus de la case non vide la plus à gauche. Une *étape de calcul* consiste à appliquer la (seule) règle applicable de la table de transition, si elle existe. Quand aucune règle ne peut s'appliquer, le calcul est fini. La sortie est le mot écrit sur le ruban à la fin du calcul.

Pour un problème de décision (réponse 1 ou 0), on définit la fin du calcul différemment. L'état du ruban à la fin du calcul n'est pas pris en compte. Par contre, on suppose que la machine possède un état spécial VRAI où aucune règle ne s'applique. Si on atteint l'état VRAI, le calcul est terminé et la réponse est 1. Si le calcul se termine dans un autre état que l'état VRAI, la réponse est 0.

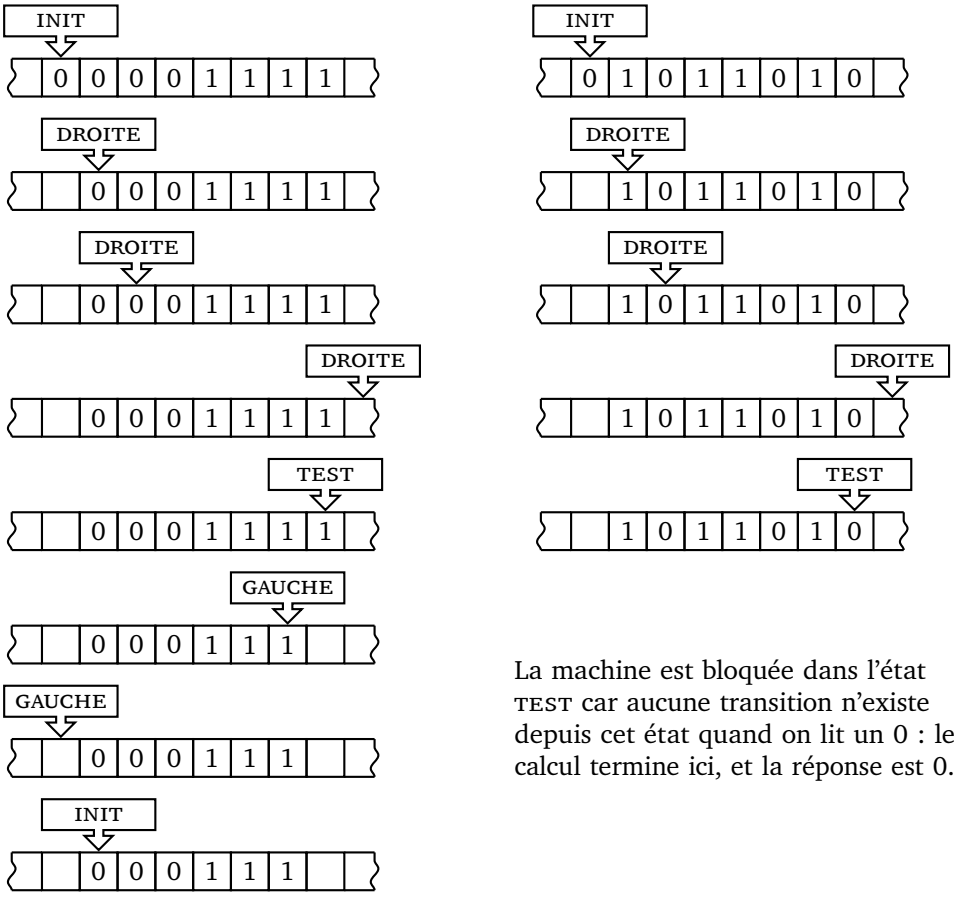
§ **Exemple** On veut calculer la fonction f qui vaut 1 quand $w = 0 \dots 01 \dots 1$ avec autant de 0 que de 1, et 0 sinon. L'algorithme qu'on utilise est le suivant : si le symbole le plus à gauche est un 1, on termine tout de suite le calcul (la réponse est 0) ; sinon on efface ce premier symbole (0) ; on déplace la tête sur le dernier symbole à droite ; si c'est un 1, on l'efface et on redéplace la tête tout gauche et on recommence ; si c'est un 0 le calcul s'arrête. Quand il n'y a plus aucun symbole sur le ruban, la machine passe dans l'état VRAI. Ainsi, le calcul termine dans l'état VRAI si on a pu effacer tous les symboles avec cette règle, et dans un autre état sinon.

Les figures 8 et 9 présentent la table de transition de cette machine et (une partie de) l'exécution de la machine sur deux entrées différentes. Dans la table, l'état DROITE est permet à la tête de se déplacer tout à droite sur le ruban, et GAUCHE permet l'inverse ; TEST efface le caractère le plus à droite si c'est bien un 1 et bloque la machine sinon (réponse 0). On a écrit *idem* dans la table quand le symbole écrit est le même que le symbole lu. Toutes les transitions possibles sont indiquées : on n'a par exemple aucune règle à appliquer dans l'état INIT en lisant un symbole 1 ; dans un tel cas, la machine s'arrête et la réponse est 0.

§ **Puissance de calcul et variantes** La machine de Turing est une des formalisations possibles de la notion d'algorithme, ce qui signifie qu'on peut effectuer exactement les mêmes calculs avec une

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT

FIGURE 8 – Table de transition d'une machine qui teste si l'entrée est de la forme $0\ldots01\ldots1$ avec autant de 0 que de 1.



La machine est bloquée dans l'état TEST car aucune transition n'existe depuis cet état quand on lit un 0 : le calcul termine ici, et la réponse est 0.

FIGURE 9 – Deux exemples d'exécution : à gauche, les premières étapes d'un calcul qui mène à VRAI ; à droite, l'entrée n'a pas la bonne forme et la machine le détecte très vite.

machine de Turing qu'avec un programme Python par exemple.

Théorème 3.2 Un problème de décision peut être calculé par une machine de Turing si et seulement si il peut l'être par un programme Python.

Pour démontrer ce résultat, il faut être capable de simuler n'importe quel calcul effectué par une machine de Turing par un programme Python, et inversement. Le sens réciproque est très technique : réussir à faire une machine de Turing qui simule un programme Python est une tâche lourde. La bonne approche, qui reste malgré tout difficile, est de passer par des langages intermédiaires plus simples. L'autre sens est beaucoup plus facile, et est l'objet de l'exercice suivant : écrire un simulateur de machine de Turing en Python.

La machine de Turing admet de nombreuses variantes, qui ne changent rien à l'ensemble des fonctions calculables. On peut lui fournir deux rubans et deux têtes de lecture/écriture, voire un nombre constant > 2 de rubans (à chaque étape, on lit et écrit un symbole par ruban), voire utiliser un ruban bi-dimensionnel. On peut également restreindre le ruban pour qu'il soit infini uniquement vers la droite. Et on peut également enrichir l'ensemble des symboles, c'est-à-dire permettre au(x) ruban(s) de contenir les symboles d'un *alphabet* fini Σ quelconque, comme les caractères ASCII. On peut également autoriser la tête de lecture à rester sur place lors d'une transition, plutôt que la forcer à bouger à droite ou à gauche. Certaines de ces variantes ont des conséquences en théorie de la complexité, mais aucune n'en a en calculabilité.

Exercice 3.1. On veut écrire un simulateur de machine de Turing en Python. Pour représenter le ruban, on utilise une liste de `0`, `1` et `-1` (pour le symbole $\square$). On représente les états par des chaînes de caractères, avec en particulier l'état `"init"` et l'état `"vrai"`. La table de transition est un dictionnaire, dont les clefs sont des couples `(etat_courant, symbole_lu)` et les valeurs des triplets `(symbole_ecrit, deplacement, nouvel_etat)`. On représente les déplacements gauche et droit par les entiers -1 et 1. Le ruban de la machine de Turing est potentiellement infini : pour simuler ce comportement, on doit agrandir le ruban (à droite ou à gauche) quand on sort de la liste qui le représente.

1. Écrire le dictionnaire correspondant à la table de l'exemple précédent.
2. Écrire une fonction `transition` qui prend en entrée une table (dictionnaire), et la configuration de la machine (état courant, ruban et position de la tête), et effectue la transition en mettant à jour le ruban, et en renvoyant le nouvel état et la nouvelle position de la tête. Si aucune transition n'est possible, la fonction renvoie `None`.
3. Écrire une fonction `simulateur` qui prend en entrée une table représentant une fonction f et un ruban initial contenant un mot w, et renvoie `True` si $f(w) = 1$ et `False` sinon.
4. *(bonus)* Encapsuler les fonctions précédentes dans une classe `MachineTuring`, et rajouter des fonctions d'affichage pour représenter le calcul en train de se faire par des images animées.

Résumé

— La machine de Turing est un dispositif *mécanique* de calcul, qu'il est possible de réaliser entièrement en Lego : `https://www.dailymotion.com/video/xrn0yi`.
— Une fonction est calculable par un programme Python si et seulement si elle l'est par une machine de Turing.
— La machine de Turing fait partie des formalismes qui ont permis l'avènement de la calculabilité, puis de l'informatique, à partir des années 1930.

3.3 Complexité

La théorie de la complexité est apparue dans les années 1960, pour comprendre quels problèmes pouvaient être résolus efficacement sur un ordinateur. En particulier, certains problèmes semblent ne pas admettre d'algorithme rapide. Les questions qui se posent alors sont : peut-on prouver qu'il n'existe pas d'algorithme rapide pour tel ou tel problème ? peut-on comprendre *pourquoi* il n'y a pas d'algorithme rapide ? La théorie de la complexité ne s'intéresse qu'aux problèmes *calculables*, et elle peut donc être vue comme un raffinement de la théorie de la calculabilité.

Les réponses apportées par la théorie de la complexité viennent par la *classification* des problèmes en *classes de complexité*, et l'étude des relations entre ces classes. Beaucoup de ces relations ne sont que conjecturales, à l'image de la célèbre question « P = NP ? ».

Comme dans le cas de la calculabilité, la théorie de la complexité nécessite un modèle fixé d'algorithme. En fait, n'importe quel modèle, qu'il soit réaliste ou absolument pas, peut donner lieu à *une* théorie de la complexité. *La* théorie de la complexité *standard* utilise la *machine de Turing à plusieurs bandes* ou de manière équivalente la *machine RAM* (voir 3.4.2). On va continuer d'utiliser les programmes Python comme définition d'algorithme. On définit la complexité en temps d'un algorithme en terme de *nombre d'opérations élémentaires* effectuées [16]. Afin de ne pas alourdir la présentation, on reste volontairement flou sur ce que sont ces opérations élémentaires, qui peuvent être des opérations sur des (petits) entiers, sur des bits, … Cependant, tous les résultats énoncés restent corrects et peuvent être démontrés dans un modèle plus formel type machine de Turing.

3.3.1 Complexité des problèmes, classes P et NP

On se restreint dans toute la discussion aux problèmes de décision, c'est-à-dire aux problèmes qui admettent une réponse binaire. Il est possible d'étudier d'autres types de problèmes en théorie de la complexité, mais cela rajoute des difficultés. En particulier les classes P et NP présentées ici concernent uniquement les problèmes de décision.

§ **Complexité des problèmes** La *complexité en temps d'un algorithme* est définie par le nombre d'opérations élémentaires qu'il effectue. La théorie de la complexité ne s'intéresse en fait (quasiment) pas à la complexité des algorithmes. Son objet d'étude est, comme pour la calculabilité, les problèmes. Cela mène à la définition de la complexité (en temps) d'un problème.

> **Définition 3.3** La *complexité (en temps) d'un problème* est la complexité du meilleur algorithme qui résout le problème, c'est-à-dire la plus petite complexité possible.

R Quand on parle de complexité d'un algorithme ou d'un problème, c'est toujours une fonction de la taille de l'entrée *représentée en binaire*. Cela ne pose que rarement des problèmes, mais il faut tout de même faire attention avec les entiers. Par exemple, l'algorithme qui teste si un entier n est premier en testant tous les diviseurs possibles entre 2 et $\sqrt{n}$ effectue $O(\sqrt{n})$ opérations. Il n'est pas polynomial car la taille de l'entrée est $O(\log n)$, le nombre de bits pour écrire l'entier n. Sa complexité *en fonction de la taille de l'entrée* est donc exponentielle, plus exactement $O(2^{\ell/2})$ où ℓ est la taille de l'entrée.

§ **Exemple** Pour illustrer la différence entre complexité des algorithmes et des problèmes, considérons le tri d'un tableau par comparaisons. Attention, on sort ici momentanément du modèle qu'on s'est

16. Voir le cours *Algorithmique*, bloc 2 du DIU.

fixé en ne s'autorisant qu'une opération particulière, la comparaison, et en considérant un problème qui n'est pas un problème de décision. On sait qu'il existe de nombreux algorithmes pour trier un tableau : les tris par insertion et sélection effectuent $O(n^2)$ comparaisons, le tri fusion en effectue $O(n \log n)$, … On en déduit que la complexité du tri est *au plus* [17] $O(n \log n)$ comparaisons. En effet, rien ne nous garantit *a priori* qu'on ne puisse pas faire mieux que cette complexité. *La complexité d'un algorithme est une borne supérieure sur la complexité du problème qu'il résout.* Dans le cas du tri, il se trouve qu'on connaît également une *borne inférieure* $O(n \log n)$ comparaisons [16]. On peut donc en déduire que la complexité du problème du tri, dans ce modèle particulier, est $O(n \log n)$ comparaisons.

§ **La classe** P La classe P (« temps polynomial ») est l'ensemble des problèmes (de décision) pour lesquels il existe un algorithme polynomial. Autrement dit, c'est l'ensemble des problèmes dont la complexité est (au plus) polynomiale. Au risque de la répétition, cette classe regroupe des *problèmes* et non des *algorithmes*. Par exemple, l'algorithme de test de primalité évoqué dans la remarque précédente n'est pas polynomial mais le problème lui-même est dans la classe P car il existe un algorithme polynomial pour le résoudre [18].

Cette classe contient beaucoup de problèmes que l'on rencontre en informatique, que ce soit des problèmes sur les entiers (est-ce que m divise n ?), les tableaux (n appartient-il à T ?), les chaînes de caractères (le motif m apparaît-il dans le texte T ?), les graphes (G est-il un arbre ?), …

La classe P a été considérée par Cobham et Edmonds (1965, indépendamment) comme la classe des problèmes *faciles*, c'est-à-dire pour lesquels il existe un algorithme *efficace*. Considérer qu'un algorithme efficace est un algorithme de complexité polynomiale est discutable, mais reste une bonne première approximation.

§ **La classe** NP Si de nombreux problèmes classiques appartiennent à la classe P, il y en a aussi beaucoup qu'on ne sait pas résoudre en temps polynomial. Par exemple, on ne connaît pas d'algorithme polynomial pour les problèmes du Chemin Hamiltonien (*étant donné un graphe G, existe-t-il un chemin qui passe une fois et une seule par chaque sommet ?*), de la Somme partielle (*étant donné un tableau T d'entiers et une cible t, existe-t-il un sous-tableau de T dont la somme des éléments vaut t ?*), ou de la Correction de chaînes (*étant donné deux chaînes de caractères s_1 et s_2 et un entier k, peut-on passer de s_1 à s_2 par au plus k suppressions de lettres et échanges de lettres adjacentes ?*). Dans ces trois cas, on sait résoudre le problème en temps exponentiel, en testant toutes les possibilités.

Exercice 3.2. Écrire un algorithme pour chacun des trois problèmes précédents.

Mais ces trois exemples ont aussi une propriété importante en commun : si on nous fournit une *solution* (un chemin, un sous-tableau, ou une suite de suppressions et échanges), on peut *vérifier* si la solution est correcte, en temps polynomial. Cela mène à la définition de l'une des classes de complexité les plus importantes : la classe NP (« temps polynomial non-déterministe » [19]). Informellement, c'est l'ensemble des problèmes de décision tels qu'une entrée est *positive*, c'est-à-dire que la réponse est oui sur cette entrée, si et seulement s'il existe une *solution* (ou *certificat*) qu'on peut *vérifier* en temps polynomial.

Pour formaliser la définition, on considère des problèmes de $\{0,1\}^\star$ dans $\{0,1\}$, c'est-à-dire que les entrées sont des *mots binaires finis* (suites de bits). Ceci n'enlève aucune généralité puisque toute entrée finie peut être codé par un mot binaire.

17. La notation $O(\cdot)$ contient déjà cette notion d'*au plus*, c'est donc ici un pléonasme.
18. Algorithme AKS, dû à M. Agrawal, N. Kayal et N. Saxena, en 2002.
19. *Nondeterministic polynomial-time* en anglais, et surtout pas « non polynomial » ! Voir 3.4.6 pour l'explication du nom.

Définition 3.4 Un problème (de décision) $\Pi : \{0,1\}^\star \to \{0,1\}$ appartient à la classe NP s'il existe un algorithme polynomial V à deux entrées, tel que pour tout $x \in \{0,1\}^\star$, $\Pi(x) = 1$ si et seulement s'il existe $y \in \{0,1\}^\star$, de taille polynomiale en la taille de x, tel que $V(x,y) = 1$.

L'algorithme V est l'algorithme de vérification, et y est la solution qu'on vérifie. On utilise plutôt la terminologie *certificat* plutôt que *solution* : y est une preuve que x est une entrée positive. On impose que y soit de taille polynomiale en la taille de l'entrée : la justification est qu'il faut que le certificat soit vérifiable en temps polynomial *en la taille de l'entrée*.

Exercice 3.3. Dans le problème du CHEMIN HAMILTONIEN, l'entrée x est le graphe G. Le certificat pour une entrée positive x est une description y d'un chemin hamiltonien.

1. Justifier que y est de taille polynomiale en la taille de x.
2. Proposer un algorithme de vérification V à deux entrées x et y qui vérifie si y est un chemin hamiltonien du graphe x.

R Dans la définition de la classe NP, il y a une dissymétrie fondamentale entre entrées *positives* et *négatives*. Si une entrée est positive, il existe un certificat qui le démontre. Si l'entrée est négative, aucun certificat ne peut démontrer le contraire. On peut définir la classe symétrique de NP, pour laquelle il existe un certificat pour les entrées négatives, et aucun pour les entrées positives. Cette classe est appelée coNP, « complémentaire de NP ».

RÉSUMÉ
- La classe P est la classe des problèmes de décision que l'on peut *résoudre* en temps polynomial.
- La classe NP est la classe des problèmes de décision que l'on peut *vérifier* en temps polynomial.

3.3.2 NP-complétude et « P = NP ? »

Les problèmes de la classe NP sont ceux qu'on sait vérifier en temps polynomial. C'est en particulier le cas de tous les problèmes de la classe P, ce qui s'écrit en symboles $\mathsf{P} \subseteq \mathsf{NP}$. En effet, si on dispose d'un algorithme polynomial A pour un problème, on en déduit un algorithme de vérification V à deux entrées : sur les entrées x et y, V ignore y (le certificat), et exécute $A(x)$.

Réciproquement, si on a un algorithme de vérification, peut-on en déduire un algorithme pour le problème ? C'est exactement l'énoncé de la question « P = NP ? », posée dans les années 1970 indépendamment par Cook (1971) et Levin (1973), de chaque côté du rideau de fer, et qui est toujours ouverte aujourd'hui [20]. La plupart des informaticiens s'accordent sur le fait que la réponse est probablement non, c'est-à-dire $\mathsf{P} \neq \mathsf{NP}$: on pense qu'il existe des problèmes dans NP qui n'admettent pas d'algorithme polynomial. Un premier indice en ce sens est qu'il y a beaucoup de problèmes dans NP pour lesquels on ne connaît pas d'algorithme polynomial (c'est le cas des trois exemples de la partie précédente).

Puisque $\mathsf{P} \subseteq \mathsf{NP}$, et que les deux classes pourraient être égales, dire qu'un problème appartient à NP ne renseigne pas sur sa difficulté. La notion de NP-complétude permet justement d'identifier, parmi les problèmes de NP, ceux qui sont *les plus difficiles*. En particulier, il suffirait de disposer

20. Avis aux amateurs, l'institut Clay de mathématiques offre un prix d'un million de dollars pour la résolution de cette question.

d'un algorithme polynomial pour n'importe lequel des problèmes NP-*complets* pour démontrer que $\mathsf{P} = \mathsf{NP}$. La conviction que $\mathsf{P} \neq \mathsf{NP}$ vient du fait qu'on connaît des dizaines de milliers de problèmes NP-complets, dans tous les domaines de l'informatique, de natures extrêmement diverses, et qu'on n'a jamais trouvé d'algorithme polynomial pour le moindre d'entre eux.

§ **Réductions polynomiales** Pour définir la NP-complétude, on a besoin de la notion de réduction entre problèmes. Si Π_1 et Π_2 sont deux problèmes de décision, une *réduction polynomiale* de Π_1 à Π_2 est un algorithme R de complexité polynomiale, qui a les mêmes entrées que Π_1 et qui pour tout x, renvoie une entrée y de Π_2 telle que $\Pi_1(x) = 1$ si et seulement si $\Pi_2(y) = 1$. Ainsi, si on dispose d'un algorithme polynomial A_2 pour Π_2, on peut déduire un algorithme polynomial pour Π_1 : `def A1(x): return A2(R(x))`.

E On rappelle que le problème Somme partielle consiste à déterminer, étant donné un tableau T d'entiers et une cible t, s'il existe un sous-tableau de T dont la somme des éléments vaut t. Le problème Partition consiste lui à déterminer, étant donné un tableau T d'entiers, s'il est possible de partitionner T en deux sous-tableaux T_1 et T_2 tels que la somme des éléments de T_1 est égale à la somme des éléments de T_2. On va exhiber une réduction de Partition à Somme partielle.
La réduction R doit avoir les mêmes entrées que le problème de départ, Partition, c'est-à-dire un tableau T d'entiers. L'objectif est de produire une entrée de Somme partielle. L'idée est la suivante : on calcule la somme s de tous les éléments de T, et on renvoie $(T, s/2)$.

```
def R(T):
    s = 0
    for t in T: s += t
    return (T,s/2)
```

Alors si T est une entrée positive de Partition, on peut séparer T en T_1 et T_2 tels que la somme des éléments de T_1 soit égale à la somme des éléments de T_2. Donc la somme des éléments de ces deux sous-tableaux sera bien $s/2$. Ce qui signifie que $(T, s/2)$ est une entrée positive de Somme partielle, puisque T_1 et T_2 sont deux solutions possibles.
Si T est une entrée négative de la Partition, il n'y a aucun moyen de trouver de tels T_1 et T_2 dont les sommes des éléments sont égales. En particulier, il n'existe aucun sous-tableau de T dont la somme des éléments vaut $s/2$. Donc $(T, s/2)$ est une entrée négative Somme partielle.
La réduction R est clairement de complexité polynomiale, et elle envoie des entrées de Partition sur des entrées de Somme partielle, de telle sorte que T est une entrée positive si et seulement si $R(T) = (T, s/2)$ en est une.

§ NP-**complétude** Une réduction polynomiale d'un problème Π_1 à Π_2 montre que Π_1 *n'est pas plus difficile que* Π_2 : en effet, si on sait résoudre Π_2 on sait aussi résoudre Π_1, *via* la réduction. Par contre, ça ne dit rien dans l'autre sens, Π_2 peut être beaucoup plus difficile à résoudre que Π_1 ou à peu près de la même difficulté.

> **Définition 3.5** Un problème de décision Π est NP-difficile si pour tout problème $\Pi' \in \mathsf{NP}$, il existe une réduction polynomiale de Π' à Π (« aucun problème de NP n'est plus difficile que Π »).

Un problème de décision est NP-complet s'il est dans NP et NP-difficile.

La NP-complétude identifie les problèmes les plus difficiles de NP. Si l'un d'entre eux est dans P, c'est que tout NP admet des algorithmes polynomiaux !

Lemme 3.6 S'il existe un problème NP-difficile Π pour lequel on dispose d'un algorithme polynomial ($\Pi \in \mathsf{P}$), alors $\mathsf{P} = \mathsf{NP}$.

Démonstration. Soit Π' un problème de NP. Puisque Π est NP-difficile, il existe une réduction polynomiale R de Π' à Π. D'autre part, Π a un algorithme polynomial A. Donc l'algorithme suivant résout Π' en temps polynomial : sur une entrée x, on applique R pour obtenir une entrée de Π, puis on applique A à $R(x)$ pour obtenir la réponse. Puisqu'on est parti de n'importe quel problème de NP, $\mathsf{P} = \mathsf{NP}$. □

Il paraît compliqué de montrer qu'un problème Π est NP-difficile : *a priori*, il faut réussir à trouver une réduction de chaque problème de NP à Π. En fait, il suffit de montrer que Π est plus difficile qu'un des problèmes les plus difficiles.

Lemme 3.7 S'il existe une réduction polynomiale d'un problème NP-difficile à Π, alors Π est NP-difficile.

Démonstration. Soit Π' un problème NP-difficile et R une réduction polynomiale de Π' à Π. Pour tout problème $\Pi'' \in \mathsf{NP}$, il existe une réduction polynomiale R' de Π'' à Π'. Donc sur une entrée x de Π'', $R(R'(x))$ produit une entrée de Π, positive si et seulement si x l'est. Donc $R \circ R'$ est une réduction polynomiale de Π'' à Π. Puisque c'est valable pour tout $\Pi'' \in \mathsf{NP}$, Π est NP-difficile. □

§ **Théorème de Cook-Levin** Grâce au lemme précédent, si on connaît un problème NP-difficile, on peut s'en servir pour montrer que d'autres problèmes le sont également. Mais il en faut bien un premier ! L'existence de problèmes NP-difficiles ne va pas de soi, et encore moins celle de problèmes NP-complets.

Théorème 3.8 Il existe des problèmes NP-complets.

Démonstration. On définit le problème AlgoSat : étant donné un algorithme de vérification V et une entrée x, existe-t-il un certificat y de taille polynomiale en la taille de x tel que $V(x, y) = 1$? Ce problème très artificiel a été fait pour être NP-complet.

Il appartient à NP, car un certificat pour AlgoSat est simplement le certificat y de l'énoncé du problème, et que l'algorithme de vérification d'AlgoSat consiste simplement à exécuter l'algorithme V sur x et y.

D'autre part, il est NP-difficile. En effet, tout problème $\Pi \in \mathsf{NP}$ admet par définition un algorithme de vérification V. La réduction à AlgoSat consiste simplement, pour une entrée x, à renvoyer (V, x). Le fait que x soit une entrée positive si et seulement si (V, x) en est une également est simplement la définition du problème AlgoSat !

Le point crucial dans cette preuve est que V ne dépend pas de x et est fixé. D'une part, la taille de y est polynomiale en la taille de x si et seulement si elle l'est en la taille de (V, x) puisque V étant fixé, sa taille est une constante. D'autre part, la question de savoir *comment calculer V à partir de x* n'a pas de sens : on écrit V « en dur » dans l'algorithme de réduction ! □

Cette preuve peut paraître tout à fait artificielle… car elle l'est. D'ailleurs, il y a un point passé sous silence. L'algorithme de vérification d'AlgoSat exécute V sur les entrées x et y. Pour que cet algorithme soit polynomial, il faut qu'on soit capable, à partir du code de V, de l'exécuter à peu près à la même vitesse que l'algorithme V lui-même. Ce point technique est un rafinement de l'existence du programme universel en calculabilité, *cf.* 3.1.2 : il existe un programme universel qui exécute n'importe quel algorithme A sur n'importe quelle entrée x, en temps $O(t(n)\log(t(n)))$ si $A(x)$ s'exécute en temps $O(t(n))$.

Cook (1971) et Levin (1973) ont montré le résultat avec un problème plus intéressant, Sat, qu'on ne décrit pas ici. Très vite après l'article de Cook, Karp (1972) a démontré que 21 problèmes étaient NP-complets par réduction depuis Sat. Parmi ces problèmes, on trouve Somme partielle, Partition et Chemin Hamiltonien. Ces 21 problèmes ont permis le développement très rapide de la NP-complétude, car ils ont fourni beaucoup de *points de départ* pour des réductions polynomiales.

En pratique

De nombreux problèmes qu'on veut résoudre en pratique sont NP-complets. Les algorithmiciens cherchent alors des solutions : algorithmes exponentiels *pas trop lents*, algorithmes efficaces *en pratique sur certains cas*, algorithmes efficaces qui résolvent une version *approchée* du problème, …
Toute la cryptographie moderne (RSA, …) repose sur la théorie de la complexité. En particulier, les protocoles cryptographiques utilisent le fait qu'il existe des fonctions *faciles à calculer* et d'autres *difficiles à calculer*.

Résumé

- Il existe des problèmes NP-complets, qui sont *les plus difficiles* de NP.
- Si n'importe lequel d'entre eux admet un algorithme polynomial, alors P = NP.
- Une réduction polynomiale de Π_1 à Π_2 permet de dire que Π_2 est plus difficile que Π_1 :
  - si Π_1 est NP-difficile, alors Π_2 également ;
  - inversement, si Π_2 est dans P, alors Π_1 également.

3.4 *Pour aller plus loin*

Cette partie présente à titre *culturel* d'autres notions bien plus avancées. Pour aller encore plus loin, je recommande les ouvrages mentionnés en introduction.

3.4.1 Dénombrabilité

Jusqu'à présent, on a parlé d'algorithmes et de fonctions, sans se pencher précisément sur les ensembles d'entrées possibles. De fait, tous les objets qu'on manipule en informatique sont représentables dans un ordinateur sous forme de mot binaire fini, c'est-à-dire d'une suite finie de bits. Cela restreint les ensembles d'objets sur lesquels on peut calculer, et est lié à la notion mathématique d'ensemble dénombrable.

Définition 3.9 Un ensemble est dénombrable s'il est possible de représenter chaque élément de l'ensemble par un mot binaire fini, de telle sorte qu'un mot représente (au plus) un élément.

On démontre facilement que cette définition est équivalente[21] à la définition standard d'ensemble dénombrable comme ensemble en bijection avec une partie de $\mathbb{N}$. Deux objets distincts doivent avoir deux représentations distinctes, mais un même objet peut éventuellement avoir plusieurs représentations et certains mots peuvent ne représenter aucun objet.

Puisque chaque caractère ASCII est représenté par 7 bits, la définition précédente est équivalente à demander que chaque élément de l'ensemble ait une représentation finie avec des caractères ASCII. On en déduit très facilement par exemple que l'ensemble des entiers est dénombrable car la représentation décimale d'un entier est une chaîne ASCII finie. De même, l'ensemble des nombres rationnels (fractions p/q) est dénombrable puisqu'on peut représenter un rationnel avec deux entiers séparées par une barre de fraction. On remarque que dans ces deux cas, chaque objet a une infinité de représentations (`5 = 05 = ...` et `3/4 = 6/8 = ...`), et que la plupart des chaînes ASCII ne représentent aucun objet de ces ensembles (`un`, `trois-quart`, `1/0`, ...).

Une grande découverte de la fin du XIX$^{\text{ème}}$ siècle (Cantor, 1874) est l'existence de plusieurs *tailles* d'infini : tous les ensembles infinis ne sont pas dénombrables. C'est par exemple le cas des nombres réels car il faut, intuitivement, une infinité de décimales pour représenter un nombre réel quelconque.

On va le démontrer pour un autre ensemble, qui nous intéresse plus directement ici. On note $\{0,1\}^\star$ l'ensemble des mots binaires finis (dont le mot *vide* de longueur nulle) et on s'intéresse à l'ensemble des fonctions (totales) de $\{0,1\}^\star$ dans $\{0,1\}$. Chaque fonction $f : \{0,1\}^\star \to \{0,1\}$ peut être représentée par un mot binaire *infini*, en concaténant les valeurs de f sur chaque mot fini (par longueur croissante puis ordre lexicographique par exemple) : $f()$, $f(0)$, $f(1)$, $f(00)$, $f(01)$, ... Par exemple, la représentation de la fonction qui vaut 1 si le nombre de 1 dans le mot binaire w est impair et 0 sinon est le mot infini `00101100...`

On démontre le résultat intuitif qu'une telle fonction n'admet pas de représentation finie en général.

Théorème 3.10 L'ensemble des fonctions $f : \{0,1\}^\star \to \{0,1\}$ est indénombrable.

Démonstration. Supposons que l'ensemble des fonctions de $\{0,1\}^\star$ dans $\{0,1\}$ soit dénombrable, c'est-à-dire que chaque fonction soit représentable par un mot *fini* w. On note f_w la fonction représentée par le mot w. On remarque qu'on peut appliquer f_w à n'importe quel mot, et donc en particulier à w.

On définit alors une fonction d, pour *diagonale*, de la façon suivante : $d(w) = 1$ si $f_w(w) = 0$ et $d(w) = 0$ sinon. Alors quelque soit w, les fonctions d et f_w sont différentes puisqu'elles n'ont pas la même valeur sur le mot w. Donc aucun mot fini w ne représente d. L'hypothèse de départ est donc contredite, et l'ensemble des fonctions de $\{0,1\}^\star$ dans $\{0,1\}$ est indénombrable. □

On peut effectuer la même preuve pour les fonctions partielles en définissant $d(w) = 1$ si $f_w(w) = 0$ ou $f_w(w) = \perp$ (indéfinie), et $d(w) = 0$ sinon. Ainsi, l'ensemble des problèmes de décision est indénombrable.

Exercice 3.4. On veut démontrer que l'ensemble des nombres réels est indénombrable, avec une preuve un peu différente que la preuve classique (pour celles et ceux qui la connaissent).

21. Mais bien plus pratique !

1. En utilisant l'écriture binaire d'un réel, montrer qu'on peut associer à chaque fonction $f : \{0,1\}^\star \to \{0,1\}$ un réel unique entre 0 et 1. *Indication. Utiliser la représentation de f par un mot infini.*
2. En déduire que l'ensemble $[0,1]$ des nombres réels entre 0 et 1 est indénombrable.
3. En déduire que c'est le cas de l'ensemble $\mathbb{R}$.

RÉSUMÉ

— Un ensemble est dénombrable si chaque élément peut être représenté par un mot binaire fini.
— Il existe des ensembles indénombrables (ensembles des fonctions de $\{0,1\}^\star$ dans $\{0,1\}$, ensembles des nombres réels, ...).
— Les objets manipulés en informatique appartiennent à des ensembles dénombrables.

3.4.2 Autres formalisations de la notion d'algorithme

En 1936, trois formalismes différents ont été proposés pour les fonctions calculables. La machine de Turing est un dispositif mécanique (théorique) qui permet d'effectuer des calculs. Elle définit donc un ensemble de *fonctions calculables par machine de Turing*. La même année, Church a proposé le λ-calcul, qui est une sorte de langage de programmation minimal (voir ci-dessous), et qui définit un ensemble de *fonctions λ-calculables*. Enfin, Herbrand (1931), Gödel (1934) puis Kleene (1936) ont défini l'ensemble des *fonctions μ-récursives*, à l'aide de fonctions de bases « évidemment calculables » (constantes, ...) et d'un ensemble de règles pour les combiner (composition, ...), voir 3.4.4. Ces auteurs ont montré dès 1936 que les fonctions calculables par machine de Turing, λ-calculables ou μ-récursives sont les mêmes.

Le fait que ces trois définitions, de natures très différentes (*mécanique* pour la machine de Turing, *syntaxique* pour le λ-calcul et *sémantique* pour les fonctions μ-récursives) donnent exactement lieu au même ensemble de fonctions calculables apporte la conviction qu'il s'agit de la *bonne* définition. On appelle ceci la *thèse de Church-Turing* : une fonction est « effectivement calculable » (par un dispositif physique) si et seulement si elle admet un algorithme (au sens de l'une des trois définitions précédentes).

D'autres définitions ont été données par la suite, qui se sont toujours avérées équivalentes aux précédentes, ce qui a renforcé la thèse de Church-Turing. Par nature, cette thèse n'est pas démontrable mathématiquement puisqu'elle est un postulat sur ce qui est physiquement réalisable. Elle pourrait être réfutée si on découvrait une manière radicalement nouvelle de faire des calculs [22].

Parmi les autres définitions, on compte tous les langages de programmation *raisonnables* [23]. La *machine RAM* (*Random Access Machine*, Cook et Reckhow, 1973) est une autre définition théorique qui permet de faire le pont avec les *vrais* langages de programmations. Elle est constituée d'un ensemble de *registres* (qui peuvent contenir chacun un symbole, disons 0 ou 1) et d'un jeu d'instructions possibles, comme par exemple effectuer le « ou logique » des symboles contenus dans deux registres et mettre le résultat dans un troisième. Un programme est simplement une suite d'instructions. La machine RAM ressemble beaucoup aux langages *assembleurs* dont on dispose sur les ordinateurs. Comme pour la machine de Turing, la définition exacte des symboles autorisés et des instructions autorisées n'a pas grande importance.

22. Je précise que le calcul quantique par exemple ne fait pas sortir de la thèse de Church-Turing : tout ce qu'on peut faire avec un ordinateur quantique peut être simulé sur un ordinateur classique. La différence qui peut exister se trouve au niveau du temps de calcul, et donc au niveau des classes de complexité.

23. Formellement, on dit *Turing-complet*. Quasiment aucun langage n'est Turing-*in*complet, puisque même le langage CSS ou LaTeX sont complets !

Parmi les définitions possibles un peu plus exotiques, il y a différents systèmes de calcul à base d'ADN, le célèbre *jeu de la vie* de John Conway, les automates cellulaires unidimensionnels, ou certains jeux vidéos comme *Minecraft*.

§ **Le λ-calcul** Le mot-clef `lambda` de Python permet de définir une fonction sans lui donner de nom. Par exemple, `lambda x,y:x+y` est la fonction *anonyme* qui fait la somme de ses entrées.

Ce mot-clef n'a pas été choisi par hasard, car cette façon de définir des fonctions *anonymes* est directement inspirée du λ-calcul. La fonction précédente s'écrit tout simplement $\lambda x \lambda y.x + y$ en λ-calcul. On remarque qu'en mathématiques, on pourrait décrire la même fonction avec la notation $(x, y) \mapsto x + y$. Le symbole λ n'a aucune signification particulière, autre qu'indiquer où se trouvent les paramètres de la fonction [24].

Ce qu'assure l'équivalence du λ-calcul avec les autres définitions d'algorithmes est que toute fonction calculable peut être écrite en λ-calcul. On peut traduire ceci en Python : tout programme Python peut s'écrire avec le mot-clef `lambda`, sans utiliser `def`, `while` ou `for` ou les appels récursifs [25]. Dans ce cadre, le programme universel est encore plus simple.

```
lambda algo,*args: exec(f'({algo}){args}')
```

Et pour se convaincre qu'on peut tout programmer de cette manière, la définition ci-dessous est une définition de l'algorithme d'Euclide du calcul de PGCD. On peut vérifier qu'`euclide(35,49)` renvoie bien 7.

```
euclide = (lambda f:(lambda x: x(x))(lambda y: f(lambda u,v: y(y)(u,v)))) \
    (lambda g:lambda a,b: a if b==0 else g(b,a%b))
```

En pratique

Différentes formalisations ont donné lieu à différentes familles de langages de programmation. En particulier, le λ-calcul est à l'origine de la programmation *fonctionnelle*, qui regroupe des langages tels que Lisp, OCaml, Haskell, Rust, Scala…

Résumé

- Les trois formalisations très différentes de la notion d'algorithme proposées en 1936, ainsi que toutes les formalisations proposées depuis, dont les langages de programmation, sont parfaitement équivalentes.
- La thèse de Church-Turing affirme que tout calcul physiquement réalisable peut être simulé par un algorithme, au sens de n'importe laquelle de ces définitions.

3.4.3 Autres problèmes incalculables

L'incalculabilité du problème de l'arrêt n'est pas un phénomène isolé. On peut définir beaucoup de fonctions non-calculables explicites.

24. D'ailleurs, cette notation est l'évolution d'une notation $\hat{x}f(x)$ utilisée par Russell et Whitehead dans les *Principia Mathematica*, devenue $\hat{}xf(x)$ chez Church, puis enfin $\lambda x.f(x)$.

25. De même on n'utilise pas le *branchement conditionnel* (`if ...: / else: ...`), mais uniquement l'*instruction conditionnelle* `x = ... if ... else ...`.

§ **Théorème de Rice** Turing avait déjà remarqué dans son article de 1936 que puisqu'on ne peut décider l'arrêt d'un programme, beaucoup d'autres caractéristiques d'un programme ne sont pas non plus calculables. Ceci est formalisé dans le théorème de Rice (1953). Étant donné le code d'un algorithme A, on peut bien sûr déterminer si l'algorithme contient une boucle `while`, ou d'autres propriétés *syntaxiques*. Mais ce que le théorème de Rice interdit, c'est de déterminer une propriété *sémantique*, c'est-à-dire une propriété du problème résolu par l'algorithme : est-ce que l'algorithme répond toujours `True` ? est-ce que l'algorithme termine sur l'entrée `"0"` ? est-ce qu'il existe au moins une entrée sur laquelle l'algorithme ne boucle pas ?

Si A est un algorithme (chaîne de caractères représentant un programme Python), on note ϕ_A la fonction que cet algorithme calcule. La notion de *propriété sémantique* signifie que la propriété porte sur ϕ_A et non sur A lui-même.

Théorème 3.11 Soit P une fonction de l'ensemble des fonctions partielles dans $\{0,1\}$, *non triviale* : il existe au moins un algorithme F tel que $P(\phi_F) = 0$ et un algorithme V tel que $P(\phi_V) = 1$. Le problème suivant est incalculable : étant donné un algorithme A, calculer $P(\phi_A)$.

On insiste sur un point : quelque soit la fonction P, le problème est non-calculable ! Même la propriété la plus simple qu'on puisse imaginer n'est pas calculable.

Démonstration. On fixe une fonction P non triviale quelconque. On va montrer qu'à partir d'un algorithme pour cette fonction P, on peut déduire un algorithme pour le problème de l'arrêt. Puisque ce dernier est incalculable, on arrive à la conclusion qu'il ne peut pas exister d'algorithme pour P.

Supposons donc qu'il existe un algorithme `propriete` tel que `propriete(A)` renvoie `True` si $P(\phi_A) = 1$ et `False` sinon. On définit l'algorithme `boucle` qui boucle sur toute entrée (on peut utiliser `while True: pass` pour cela). On suppose que $P(\phi_{\texttt{boucle}}) = 0$. Le cas $P(\phi_{\texttt{boucle}}) = 1$ est similaire, en considérant la fonction $\neg P$ qui vaut 1 si P vaut 0 et inversement.

On définit un algorithme pour le problème de l'arrêt comme suit.

```
def arret(A, m):
    algo = """
        def f(n):
            x = universel(A,m)
            return universel(V,n)
        """
    return propriete(algo)
```

Supposons que l'algorithme A s'arrête sur l'entrée m. Alors l'algorithme `algo` calcule exactement la même fonction que `V` puisqu'il commence par calculer $x = A(m)$, ignore le résultat, puis calcule $V(n)$. Autrement dit, $\phi_{\texttt{algo}} = \phi_V$. Mais si A ne s'arrête pas sur l'entrée m, `algo` ne s'arrête sur aucune entrée puisqu'il ne dépasse jamais la ligne `x = universel(A,m)`. Dans ce cas, $\phi_{\texttt{algo}} = \phi_{\texttt{boucle}}$. Puisque $P(\phi_V) = 1$ et $P(\phi_{\texttt{boucle}}) = 0$, déterminer la valeur de $P(\phi_{\texttt{algo}})$ permet de savoir si $\phi_{\texttt{algo}} = \phi_V$ ou si $\phi_{\texttt{algo}} = \phi_{\texttt{boucle}}$, donc de déterminer si A s'arrête sur l'entrée m. Donc `propriete` permet de résoudre le problème de l'arrêt. Or c'est impossible, donc `propriete` n'existe pas. □

En pratique

Le théorème de Rice permet par exemple de conclure qu'un compilateur parfait ne peut pas exister : un compilateur ne pourra jamais prévenir l'utilisateur à tous les coups que son programme boucle par exemple. Bien sûr, cela n'empêche pas qu'un compilateur puisse *parfois* prévenir d'un tel comportement, quand le comportement est prévisible syntaxiquement. Une autre conséquence est qu'il est impossible, en général, de vérifier algorithmiquement si deux programmes font la même chose ou si un programme satisfait les spécifications qu'on lui a fixées.

§ **Théorème de Matiyasevich** Au delà des questions liées aux fonctions calculables comme dans le théorème de Rice, le phénomène d'incalculabilité peut être *observé* pour des problèmes naturels. Par exemple le 10ème problème de Hilbert est le suivant : étant donné équation polynomiale à coefficients entiers, décider si elle admet une solution entière. Par exemple $3x^2 - 2xy + xz^7 - 12 = 0$ admet $x = y = 3$ et $z = 1$ comme solution entière.

Matiyasevich (1970) a démontré qu'il n'existe pas d'algorithme pour résoudre ce problème. La preuve de ce résultat est extrêmement difficile. Mais on peut tout de même donner quelques intuitions. La première, et c'est en très simplifié le contenu de la preuve de Matiyasevich, une telle équation polynomiale peut avoir une solution entière, mais avec des nombres astronomiquement grands même quand l'équation est désespérément simple. Par exemple, l'équation $x^3 + y^3 + z^3 = 42$ a bien des solutions, mais la plus petite (trouvée en 2019, on n'en connaissait pas avant) est

$$x = -80538738812075974 \quad y = 80435758145817515 \quad z = 12602123297335631.$$

La deuxième intuition est liée au fait qu'on peut *encoder* des problèmes très divers dans une telle équation. Par exemple, résoudre l'équation $a = (x+2) \times (y+2)$ (qu'on peut réécrire sous la forme $a = xy + 2x + 2y + 4$) revient à factoriser a ; résoudre l'équation $x^2 - d(y^2+1) = 1$ permet de déterminer si d est un carré parfait ; ...

En fait, ce que dit le résultat de Matiyasevitch, c'est que pour n'importe quelle fonction calculable [26] $f : \mathbb{N} \to \{0,1\}$, il existe une équation en les variables n, x_1, ..., x_k telle que l'équation a une solution si et seulement si $f(n) = 1$.

R Si on cherche l'existence d'une racine réelle, le problème devient calculable. Si on cherche l'existence d'une racine rationnelle, on n'en sait rien : on ne dispose ni d'un algorithme, ni d'une preuve d'incalculabilité !

§ **Problème de correspondance de Post** Ce problème, proposé par Post (1946), est une forme de jeu. On dispose d'un ensemble fini de *dominos*, sur lesquels sont écrits des mots : $\left[\frac{a}{ab}\right]$ est un domino avec le mot a en haut et le mot ab en bas. L'objectif est de construire une suite finie de dominos de telle sorte que les mots en haut et en bas soit les mêmes (on peut répéter chaque domino autant de fois qu'on le souhaite). Par exemple, si on a les dominos $\left[\frac{b}{ca}\right]$, $\left[\frac{a}{ab}\right]$, $\left[\frac{ca}{a}\right]$ et $\left[\frac{abc}{c}\right]$, on peut construire la suite $\left[\frac{a}{ab}\right]\left[\frac{b}{ca}\right]\left[\frac{ca}{a}\right]\left[\frac{a}{ab}\right]\left[\frac{abc}{c}\right]$ pour avoir en haut et en bas le mot $abcaaabc$. Par contre, on voit facilement qu'avec l'ensemble de dominos $\left[\frac{abc}{ab}\right]$, $\left[\frac{ca}{a}\right]$ et $\left[\frac{acc}{ba}\right]$ c'est impossible : le mot du haut sera toujours plus long que celui du bas.

26. On se restreint ici aux fonctions de $\mathbb{N}$ dans $\mathbb{N}$, mais avec un *encodage* approprié cela fonctionne pour n'importe quelle fonction calculable.

Le *problème de correspondance de Post* consiste à déterminer, étant donné un ensemble de domino, s'il est possible de construire une suite finie décrivant les mêmes mots en haut et en bas. Ce jeu tout simple est incalculable !

§ **Problème des tuiles de Wang** Ce problème ressemble vaguement au problème de correspondance de Post. On se donne un ensemble fini de pièces de puzzle (appelées *tuiles*) avec des motifs sur les bords. On dispose d'autant de copies de chaque pièce que l'on souhaite. La question qu'on se pose est de savoir s'il est possible de faire un puzzle aussi grand qu'on le souhaite (ou infini) avec ces pièces, sachant qu'il n'est pas autorisé de tourner les pièces (les directions haut/bas et gauche/droite sont fixées).

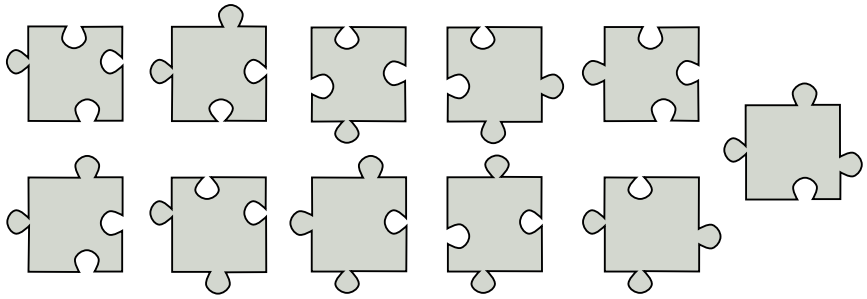


FIGURE 10 – Peut-on faire un puzzle aussi grand qu'on veut avec ces pièces ? *Ce jeu de 11 pièces est le plus petit qui admette un puzzle infini, mais aucun puzzle périodique (Jeandel & Rao, 2015).*

RÉSUMÉ

- Aucune question non triviale sur ce que calcule un algorithme ne peut être résolue par un algorithme (Rice).
- Le phénomène de non-calculabilité existe dans des domaines divers et variés.

3.4.4 Les fonctions récursives primitives et la boucle `for`

Les théorèmes de calculabilité permettent de savoir ce qui est strictement nécessaire pour un langage de programmation, et ce qui est *superflu*. On étudie cette question du point de vue des fonctions qu'on peut calculer, pas de la facilité qu'il y a à utiliser le langage…

Un premier exemple est l'utilisation d'algorithmes *récursifs* (avec des appels récursifs) ou *itératifs* (avec des boucles). On peut montrer que les deux approches sont équivalentes, ce qui veut dire qu'on peut se passer au choix des appels récursifs ou des boucles. D'ailleurs, les compilateurs suppriment de fait les appels récursifs dans les programmes et les simulent à l'aide (de l'équivalent) d'une boucle `while`, en gérant une *pile d'appels*.

§ **Récursion primitive** L'exemple qu'on va étudier plus en détail est la différence entre boucles `while` et `for`. Il est assez clair qu'on puisse remplacer n'importe quelle boucle `for` par une boucle `while` en gérant soi-même l'indice de boucle. Il est également relativement intuitif que l'inverse n'est pas toujours possible [27]. On va le démontrer grâce au formalisme des *fonctions récursives primitives* : une

27. Attention : il faut ici considérer les *vraies* boucles `for` comme en Python dans lesquels on ne peut pas modifier l'indice de boucle dans le corps de la boucle. En C par exemple, les boucles `for` sont en fait aussi puissantes que les boucles `while` car on peut faire ce qu'on veut avec l'indice de boucle.

fonction est récursive primitive si on peut la calculer à l'aide d'un programme Python qui ne contient que des boucles `for` et des branchements conditionnels `if ...:/else:`, à l'exclusion des boucles `while`, des appels récursifs et des `lambda`. Historiquement, ces fonctions ont été introduites comme tentative de formalisation de la notion d'algorithme, et elles ont ensuite donné lieu au formalisme des fonctions μ-récursives (voir ci-dessous).

La première remarque est que les fonctions récursives primitives sont totales (définies partout). En effet, sans appel récursif et sans boucle `while`, aucun risque de boucler. Un langage de programmation qui ne permettrait que les fonctions récursives primitives aurait donc l'énorme avantage de n'avoir aucun programme qui boucle. Malheureusement, certaines fonctions qu'on aimerait pouvoir programmer ne seraient pas programmables dans ce cas comme on va le voir. Ceci justifie l'importance des fonctions partielles, même pour ne calculer que des fonctions totales : aucun langage de programmation ne permet à la fois de programmer toutes les fonctions totales, et uniquement des fonctions totales.

§ **Fonctions calculables non récursives primitives** Pour démontrer qu'il existe des fonctions totales non récursives primitives, on utilise l'idée du programme universel. On va montrer que le programme universel des fonctions récursives primitives n'est pas lui-même récursif primitif. On rappelle qu'un *programme universel* n'est rien d'autre qu'un interpréteur. Bien sûr, il existe un interpréteur calculable pour les fonctions récursives primitives. En effet, les fonctions récursives primitives sont un sous-ensemble des fonctions calculables, donc l'interpréteur des fonctions calculables (le programme `universel` défini précédemment) est un interpréteur valable pour les fonctions récursives primitives. On peut le modifier pour qu'il n'accepte que les fonctions récursives primitives : il suffit de tester s'il y a une boucle `while` et s'il y a des appels récursifs. Ainsi modifié, on obtient un interpréteur pour les fonctions récursives primitives, qui ne boucle jamais car les fonctions récursives primitives ne bouclent pas.

Pour montrer qu'il n'existe pas d'interpréteur des fonctions récursives primitives qui soit lui-même récursif primitif, on effectue le même type de preuve que pour le problème de l'arrêt, en utilisant de manière cruciale le fait que toute fonction récursive primitive termine sur toute entrée. On se place à nouveau dans le cadre des problèmes de décision, donc tout algorithme renvoie soit `True` soit `False`.

> **Théorème 3.12** Aucun programme universel pour les fonctions récursives primitives n'est récursif primitif.

Démonstration. On considère un programme `universel_RP` qui sur les entrées `algo_RP` et m, simule `algo_RP` sur m. On définit le programme suivant.

```
def diagonal_RP(algo_RP):
    return not universel_RP(algo_RP, algo_RP)
```

Si `universel_RP` était récursif primitif, alors `diagonal_RP` le serait aussi : il n'utilise ni boucle `while`, ni appel récursif. Si on veut n'avoir aucun appel de fonction, il suffit de copier le code de `universel_RP` dans `diagonal_RP`. Supposons donc que `diagonal_RP` soit récursif primitif. On peut alors l'appliquer à son propre code. On retrouve la même contradiction que pour le problème de l'arrêt : si `diagonal_RP(<diagonal_RP>)` renvoie `True` il doit renvoyer `False` et inversement. Ainsi, `diagonal_RP` et `universel_RP` ne sont pas primitifs récursifs. □

On peut avoir l'impression que la même preuve fonctionnerait pour les fonctions calculables, pour montrer que le programme universel n'est pas lui-même calculable. L'argument ne fonctionne pas pour les fonctions calculables, car rien n'assure que l'application du programme `diagonal` à son code termine. Si on définit `def diagonal(algo): return not universel(algo, algo)`, on est même sûr que `universel(<diagonal>, <diagonal>)` boucle.

§ **Fonctions d'Ackermann-Péter** L'une des premières fonctions qui a été montrée non récursive primitive est la fonction d'Ackermann. On présente une version plus simple due à Péter et Robinson, et appelée… fonction d'Ackermann-Péter[28]. Cette fonction étant calculable, on fournit le programme Python correspondant (qu'il est déconseillé de lancer avec des valeurs supérieures à 3…).

```
def AckPeter(m,n):
    if m == 0: return n+1
    if n == 0: return AckPeter(m-1, 1)
    return AckPeter(m-1, AckPeter(m, n-1))
```

Il est déjà non trivial que la fonction termine sur toute entrée $(m,n) \in \mathbb{N}^2$. Pour le démontrer, on définit un ordre sur les couples : $(m,n) \prec (m',n')$ si $m < m'$ ou ($m = m'$ et $n < n'$). Alors `AckPeter(m,n)` n'effectue que des appels récursifs sur des couples inférieurs à (m,n) pour l'ordre $\prec$, ce qui montre sa terminaison.

On pose $A(n) =$ `AckPeter(n,n)`. Les premières valeurs sont $A(0) = 1$, $A(1) = 3$, $A(2) = 7$, $A(3) = 61$. Mais l'appel `AckPeter(4,4)` ne termine déjà pas en pratique. On sait, par calcul à la main, que $A(4) = 2^{2^{2^{65536}}} - 3$. Remarquons que pour écrire ce nombre en binaire, il faudrait $2^{2^{65536}}$ bits. Pour rappel, l'univers observable contient à peine $5 \times 10^{81} \simeq 2^{272}$ atomes. Le nombre $A(4)$ n'est donc pas astronomique, il est beaucoup plus gros que cela ! C'est d'ailleurs de cette manière qu'on peut montrer qu'`AckPeter` n'est pas récursive primitive : elle croît plus vite que n'importe quelle fonction récursive primitive.

§ **Définition historique** La définition historique justifie le nom de fonctions récursives primitives. On se restreint, dans cette définition, aux algorithmes qui prennent en entrées des entiers naturels et dont la sortie est aussi un entier naturel. On commence par définir des *fonctions de base* : la fonction constante nulle, la fonction successeur et les projections (`proj_n_k` existe pour tout n et k).

```
def zero():          def succ(n):            def proj_5_3(x1,x2,x3,x4,x5):
    return 0             return n + 1            return x3
```

Ensuite, on définit deux règles de combinaisons. Si on dispose d'une fonction `f` à k entrées et de k fonctions `g1`, …, `gk` à t entrées, on peut définir leur *composition* :

```
def f_g1_gk(x1,...,xt):
    return f(g1(x1,...,xt), ..., gk(x1,...,xt))
```

Si on dispose de deux fonctions `f` et `g` à k et $k+2$ entrées respectivement, on peut définir la fonction `rec_f_g` par *récursion primitive* sous la forme suivante :

28. Non ce n'est pas très logique. Remarquons cependant que pour une fois, c'est le nom de l'auteur (Raphael M. Robinson) qui est oublié et celui de l'autrice (Rósza Péter) qui est conservé. Rareté remarquable…

```
def rec_f_g(n, x1, ..., xk):
    if n == 0: return f(x1, ..., xk)
    return g(x1,...,xk, n-1, rec_f_g(n-1, x1,...,xk))
```

On obtient les fonctions récursives primitives en se restreignant aux programmes obtenus à partir des fonctions de bases, à l'aide de ces deux règles, *à l'exclusion de tout autre construction du langage*. Par exemple, l'addition et la multiplication sont récursives primitives selon cette définition :

```
def plus(m, n):
    if m == 0: return proj_1_1(n)
    return succ(proj_3_3(n,m-1,plus(m-1,n)))
```

```
def fois(m, n):
    if m == 0: return zero()
    return plus(n,fois(m-1,n))
```

Exercice 3.5.

1. Identifier les fonctions `f` et `g` telles que `plus = rec_f_g`.
2. Le programme `fois` ne respecte pas exactement la définition de la récursion primitive. En donner une version qui la respecte scrupuleusement.
3. Montrer que les fonctions fact : $n \mapsto n!$, exp : $(m,n) \mapsto m^n$ ou encore diff : $(m,n) \mapsto |m-n|$ sont récursives primitives.
4. Montrer qu'on peut simuler n'importe quelle fonction récursive primitive au sens de la définition historique par un programme n'ayant que des boucles `for` et aucun appel récursif.
   L'autre sens est un peu plus pénible, car les programmes Python qui n'utilisent que la boucle `for` peuvent tout de même être assez tordus. Cependant, les deux définitions sont bien équivalentes.

§ **Fonctions μ-récursives** Les fonctions μ-récursives, équivalentes aux machines de Turing ou tout autre formalisme de fonctions calculables, sont définies comme les fonctions primitives récursives, en ajoutant une règle supplémentaire. Si on dispose d'une fonction `f` ayant $k+1$ entrées, sa *minimisation* (ou *récursion non bornée*) `mu_f` est la fonction

```
def mu_f(x1, ..., xk):
    n = 0
    while f(n, x1, ..., xk) != 0:
        n += 1
    return n
```

La théorie de calculabilité assure qu'on peut programmer n'importe quelle fonction calculable, à l'aide des fonctions de base, de la composition, de la récursion primitive et de la minimisation.

Résumé

- Les programmes qui ne comportent ni boucles `while` ni appels récursifs sont strictement moins puissants que les algorithmes généraux.
- La formalisation de ces programmes définit les fonctions récursives primitives.
- La fonction d'Ackermann-Péter est un exemple de fonction calculable, qu'aucun ordinateur ne parviendra jamais à calculer pour des entrées ≥ 4.
- Les fonctions μ-récursives sont le troisième formalisme de la notion d'algorithme proposé en 1936.

3.4.5 Formes normales et autres curiosités

La calculabilité regorge de résultats qui, une fois traduits dans un langage de programmation comme Python, donnent des curiosités intéressantes.

Le *théorème de forme normale* assure que toute fonction calculable admet un algorithme sans appel récursif, avec une unique boucle `while` en tête de l'algorithme, et des boucles `for` et des branchements conditionnels contenus dans la boucle. Autrement, dit, tout algorithme peut s'écrire sous la forme suivante.

```
def forme_normale(<paramètres>):
    while <condition>:
        ... #contenu du programme
```

Le *théorème du point fixe* de Kleene (1938) affirme que pour toute *fonction totale calculable* σ, il existe un algorithme A tel que $\phi_A = \phi_{\sigma(A)}$. Cela se traduit sous la forme suivante : si un algorithme S prend en entrée des algorithmes et renvoie des algorithmes [29], alors il existe un algorithme A tel que A et $S(A)$ calculent la même fonction. Ce résultat a plusieurs conséquences.

— Il existe pour n'importe quel langage de programmation des *quines*, c'est-à-dire des programmes qui écrivent leur propre code.

```
def quine():
  s = 'print("def quine():\\n    s =", repr(s),"\\n   ",s)'
  print("def quine():\n    s =", repr(s),"\n   ",s)
```

— Il n'existe pas d'algorithme qui étant donné un algorithme A, trouve le plus petit algorithme (en nombre de caractères) qui calcule ϕ_A.

Le *théorème d'isomorphisme* de Roger (1967) affirme qu'il existe une bijection [30] calculable ρ qui étant donnés deux langages de programmation L_1 et L_2, envoie tout algorithme A_1 écrit dans le langage L_1 sur un algorithme $A_2 = \rho(A_1)$, de telle sorte que A_1 et A_2 calculent la même fonction. Cela démontre simplement l'existence d'un compilateur entre tout couple de langage. Cela signifie en fait plus fort : si on compile un programme C par exemple en utilisant la fonction ρ, on peut à partir de l'exécutable produit reconstruire le programme C d'origine ! Dans la réalité, les compilateurs ne permettent pas cela, mais ça serait *théoriquement* possible d'en construire avec cette propriété.

Résumé

— Le théorème de forme normale prévoit quelles formes peuvent prendre des algorithmes.
— D'autres théorèmes prévoient l'existence de certains programmes, qu'on sait ou non expliciter.

3.4.6 Le non-déterminisme

Le *non-déterminisme* du nom de la classe NP renvoie à une autre façon de définir cette classe, à l'aide d'*algorithmes non-déterministes*. Dans notre cadre, on va définir des programmes Python

29. C'est-à-dire que ses entrées et ses sorties sont des programmes exécutables, de la forme `"def ..."`.
30. Une *bijection* entre deux ensemble E et F est une fonction $f : E \to F$ inversible : il existe $f^{-1} : F \to E$ telle que pour tout $x \in E$, $x = f^{-1}(f(x))$.

non-déterministes : ces programmes ne peuvent pas être exécutés, et ne sont qu'une vue de l'esprit. On enrichit le langage à l'aide d'un nouveau symbole `$` (qui n'est pas utilisé en Python) et on remplace les affectations standard (`x = y + z`, ...) par des affectations *non-déterministes* : `x = y + z $ y - z` signifie dans notre nouveau langage « `x` reçoit, de manière non-déterministe, soit la valeur `y+z`, soit la valeur `y-z` ». Un programme non-déterministe calcule un problème Π si sur toute entrée positive, il est possible de sélectionner pour chaque affectation non-déterministe une des deux alternatives, de telle sorte que le programme non-déterministe renvoie `True`, et que c'est impossible sur les entrées négatives.

E Le programme non-déterministe suivant résout Somme partielle.

```
def somme_partielle(T, t):
    s = 0
    for i in range(len(T)):
        s += T[i] $ 0
    return s == t
```

S'il est possible de trouver un sous-tableau de T dont la somme des éléments vaut t, il suffit de sélectionner la première alternative pour ces éléments là, et la seconde pour tous les autres. La variable s vaudra t à la fin. Si par contre (T, t) est une entrée négative, quelque soient les choix effectués à chaque affectation, s ne vaudra jamais t à la sortie.

Lemme 3.13 La classe NP est égale à l'ensemble des problèmes qui admettent un algorithme non-déterministe polynomial.

Démonstration. Si Π est dans NP, d'après la définition 4, il existe un algorithme de vérification V pour Π. On le transforme en un programme non-déterministe. Sur une entrée x, le programme non-déterministe commence par *deviner* un certificat y, en faisant un nombre polynomial d'affectations non déterministes `y[i] = 0 $ 1`. Puis il exécute l'algorithme V sur l'entrée (x, y). Le programme non-déterministe peut deviner un certificat tel que $V(x, y) = 1$ si et seulement s'il en existe un, donc il calcule bien Π.

Dans l'autre sens, supposons qu'on dispose d'un programme non-déterministe pour Π. On le transforme en un algorithme de vérification. Le certificat pour une entrée positive x est la suite des choix à effectuer pour chaque affectation non-déterministe : disons que c'est une liste `C` de booléens, `True` signifiant qu'on choisit la première alternative et `False` la seconde. On modifie le programme non déterministe pour obtenir un algorithme de vérification, qui prend en entrée x et la liste des choix. Chaque affectation non-déterministe `var = val1 $ val2` est remplacée par l'affectation conditionnelle `var = val1 if C[i] else val2 ; i = i+1`, le compteur `i` étant initialisé à 0 au début de l'algorithme. Il est clair que l'algorithme obtenu est bien un algorithme de vérification pour Π. □

Résumé

— On peut définir la classe NP avec des algorithmes *non-déterministes*, qui sont des objets purement théoriques qui ne peuvent pas être exécutés.

3.4.7 Autres classes de complexité

On a décrit jusqu'à maintenant deux classes de complexité en temps, déterministe et non-déterministe. On peut définir beaucoup d'autres classes de complexité intéressantes[31]. Si on prend n'importe quelle *borne de complexité*, c'est-à-dire une fonction $T : \mathbb{N} \to \mathbb{N}$, on peut définir les classes $\mathsf{DTIME}(T(n))$ et $\mathsf{NTIME}(T(n))$ qui regroupent l'ensemble des problèmes qu'on peut respectivement *résoudre* et *vérifier* en temps $O(T(n))$. Alors $\mathsf{P} = \bigcup_k \mathsf{DTIME}(n^k)$ et $\mathsf{NP} = \bigcup_k \mathsf{NTIME}(n^k)$.

§ **Temps exponentiel** Au delà du temps polynomial, on peut définir des classes pour le temps exponentiel : $\mathsf{EXP} = \bigcup_k \mathsf{DTIME}(2^{n^k})$ est la classe des problèmes qu'on peut résoudre en temps exponentiel et $\mathsf{NEXP} = \bigcup_k \mathsf{NTIME}(2^{n^k})$ est la classe de ceux qu'on peut vérifier en temps exponentiel. On sait alors que

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}.$$

En effet, tout problème de NP peut-être résolu en temps exponentiel : il *suffit* d'essayer tous les certificats possibles (il y en a un nombre exponentiel) et d'appliquer l'algorithme de vérification avec chacun d'entre eux. Cependant, on ne sait pour aucune de ces inclusions si elle est stricte ou non. Tout ce qu'on sait dire, c'est que certaines le sont.

Théorème 3.14 On a $\mathsf{P} \neq \mathsf{EXP}$ et $\mathsf{NP} \neq \mathsf{NEXP}$.

La démonstration repose sur le fait de savoir exécuter un algorithme tout en maintenant un compteur du nombre d'opérations élémentaires effectuées, en le stoppant si ce nombre d'opérations dépasse une certaine borne. Donner un sens exact à ce résultat nécessite une formalisation plus précise des algorithmes, on l'admet donc.

Idée de la démonstration. On considère une variante du problème de l'arrêt : étant donné un algorithme A et une entrée x, est-ce que A répond OUI sur l'entrée x en effectuant au plus 2^n opérations élémentaires, où n désigne la taille de x ?

Ce problème est dans EXP : on exécute l'algorithme A sur l'entrée x, en le stoppant s'il fait plus de 2^n opérations. Si A a terminé son calcul, on répond la même chose que A. Sinon on répond NON. Cet algorithme utilise $O(2^n \log 2^n) = O(2^{n+\log n})$ opérations élementaires.

Montrons qu'il n'est pas dans P. Supposons par l'absurde qu'il existe un algorithme polynomial B qui prend en entrée A et x, et répond OUI si $A(x)$ s'arrête en $\leq 2^n$ opérations, et NON sinon. À partir de B, on crée l'algorithme D (pour « diagonal ») qui rappelle des souvenirs : D prend en entrée le code d'un algorithme A, et exécute B sur l'entrée $(\langle A\rangle, \langle A\rangle)$, où $\langle A\rangle$ représente le code de A ; puis D répond le contraire de ce que répond B. Maintenant, appliquons D à son propre code : calculer $D(\langle D\rangle)$ prend un temps polynomial (puisque ça consiste à calculer $B(\langle D\rangle, \langle D\rangle)$, et que B est polynomial). Si $D(\langle D\rangle)$ renvoie OUI, cela signifie que $B(\langle D\rangle, \langle D\rangle)$ a renvoyé NON, c'est-à-dire que D ne renvoie pas OUI sur l'entrée $\langle D\rangle$ en $\leq 2^{|\langle D\rangle|}$ étapes (où $|\langle D\rangle|$ est la taille du code de l'algorithme D). Si $D(\langle D\rangle)$ renvoie NON, cela signifie que $B(\langle D\rangle, \langle D\rangle)$ renvoie OUI, c'est-à-dire que D répond OUI sur l'entrée $\langle D\rangle$ en $\leq 2^{|\langle D\rangle|}$ opérations. Dans les deux cas, c'est absurde. Donc B n'existe pas, et le problème n'appartient pas à P.

La preuve de $\mathsf{NP} \neq \mathsf{NEXP}$ est plus subtile, à cause du non-déterminisme. □

31. On peut formellement définir une infinité non dénombrable de classes de complexité, dont la quasi-totalité n'a aucun intérêt !

§ **Complexité en espace** Quand on analyse la complexité d'un algorithme, on peut analyser l'espace mémoire utilisé. On dit qu'un algorithme a une complexité en espace $O(s(n))$ si sur une entrée de taille n, il n'utilise comme espace de travail pas plus de $O(s(n))$ bits. On peut définir des classes de complexité en espace, de manière similaire au temps : $\mathsf{DSPACE}(s(n))$ et $\mathsf{NSPACE}(s(n))$ sont les classes des problèmes qu'on peut respectivement résoudre et vérifier en espace $O(s(n))$. On remarque que dans ces classes, on ne borne pas le temps de calcul.

Les plus intéressantes en pratiques sont les classes $\mathsf{L} = \mathsf{DSPACE}(\log(n))$ et $\mathsf{NL} = \mathsf{NSPACE}(\log n)$ (espace logarithmique), et $\mathsf{PSPACE} = \bigcup_k \mathsf{DSPACE}(n^k)$ (espace polynomial). Et NPSPACE ? On peut bien sûr définir également cette classe, mais elle est égale à PSPACE. Plus généralement, le théorème de Savitch (1970) affirme que $\mathsf{NSPACE}(s(n)) \subseteq \mathsf{DSPACE}(s(n)^2)$.

Alors qu'on a défini des classes en temps polynomial et exponentiel, on définit des classes en espace logarithmique et polynomial. En effet, il est rare (mais pas impossible) qu'un problème polynomial requière plus qu'un espace logarithmique. On connaît quelques liens entre classes de complexité en temps et en espace. Le théorème suivant résume l'état des connaissances aujourd'hui.

Théorème 3.15 On a les inclusions suivantes :

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}.$$

De plus, $\mathsf{NL} \neq \mathsf{PSPACE}$.

Idée de la démonstration. On connaît déjà certaines inclusions. La preuve de $\mathsf{NL} \neq \mathsf{PSPACE}$ est proche de celle de $\mathsf{P} \neq \mathsf{EXP}$ (en utilisant le théorème de Savitch).

L'inclusion $\mathsf{P} \subseteq \mathsf{PSPACE}$ vient du fait qu'un algorithme effectuant un nombre d'opérations élémentaires polynomial ne peut pas écrire plus qu'un nombre polynomial de bits. Il utilise donc un espace mémoire polynomial au maximum. On étend ce raisonnement à l'inclusion $\mathsf{NP} \subseteq \mathsf{PSPACE}$: on peut résoudre un problème de NP en énumérant tous les certificats possibles et en exécutant l'algorithme de vérification. Puisque la vérification se fait en temps polynomial, elle s'effectue également en espace polynomial. De plus, énumérer les certificats ne rajoute qu'un espace supplémentaire polynomial : en effet, il est inutile d'écrire tous les certificats en mémoire en même temps (ce qui prendrait un espace exponentiel), on peut se contenter de les écrire les uns après les autres.

Pour montrer que $\mathsf{L} \subseteq \mathsf{P}$, on remarque que si un algorithme n'utilise que $O(\log n)$ bits d'espace supplémentaire, l'état de sa mémoire ne peut prendre que $2^{O(\log n)} = O(n^k)$ (pour un certain k) valeurs possibles. Supposons qu'à deux instants différents, l'état de la mémoire est le même. Alors l'algorithme va forcément boucler. En effet, l'état de la mémoire est le seul moyen pour l'algorithme de savoir où il en est de son calcul. Donc s'il passe deux fois par le même état de la mémoire, il y repassera indéfiniment [32]. On en déduit que le nombre d'opérations maximal effectué par un tel algorithme est $O(n^k)$, donc $\mathsf{L} \subseteq \mathsf{P}$. L'inclusion $\mathsf{PSPACE} \subseteq \mathsf{EXP}$ a exactement la même preuve. La preuve de $\mathsf{NL} \subseteq \mathsf{P}$ est un peu plus technique. □

§ **Complexité probabiliste** Certains algorithmes utilisent des tirages aléatoires, et ne rentrent pas dans le cadre étudié jusqu'ici. Formellement, on ajoute la possibilité à un algorithme d'obtenir des bits aléatoires, *via* une fonction `random_bit()` par exemple. On sait démontrer que cette fonction suffit à obtenir des entiers aléatoires entre deux bornes, à simuler une gaussienne, ... Un algorithme

32. Cet argument peut être rendu formel dans un modèle formel d'algorithmes.

probabiliste, qui utilise `random_bit`, ne renverra pas toujours le même résultat sur une entrée donnée. On dit qu'un algorithme probabiliste A a une complexité polynomiale si, quelque soient les bits aléatoires tirés, il termine toujours en temps polynomial.

De cette manière, on peut définir plusieurs classes qui correspondent toutes à *une* notion de *temps polynomial probabiliste*.

— BPP (« temps polynomial probabiliste à erreur bornée ») : $\Pi \in \mathsf{BPP}$ s'il existe un algorithme polynomial probabiliste A tel que sur toute entrée x, $A(x)$ renvoie la bonne réponse ($= \Pi(x)$) avec probabilité $\geq 2/3$;
— RP (« temps polynomial *randomisé* ») : $\Pi \in \mathsf{RP}$ s'il existe un algorithme polynomial probabiliste A qui n'effectue que des *faux négatifs*, c'est-à-dire que sur une entrée positive x, $A(x)$ renvoie 1 avec probabilité $\geq 2/3$ alors que sur une entrée négative, A renvoie toujours 0 ;
— coRP : c'est la classe symétrique de RP des problèmes qui admettent un algorithme polynomial probabiliste qui n'effectue que des *faux positifs* ;
— ZPP (« temps polynomial probabiliste sans erreur ») : $\Pi \in \mathsf{ZPP}$ s'il existe un algorithme polynomial probabiliste A qui ne se trompe jamais, mais répond « je ne sais pas » avec probabilité $\leq 1/3$.

Les classes BPP, RP et coRP correspondent aux algorithmes appelés *Monte Carlo* en algorithmique, et ZPP aux algorithmes *Las Vegas*. Il faut noter que ZPP peut être définie de manière équivalente par des algorithmes probabilistes qui répondent toujours correctement, mais dont l'*espérance* du temps de calcul est polynomiale (définition plus classique de *Las Vegas*).

Théorème 3.16 On a $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$. De plus,

$$\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \mathsf{RP} \subseteq \mathsf{BPP} \subseteq \mathsf{PSPACE}$$

et $\mathsf{RP} \subseteq \mathsf{NP}$.

Idée de la démonstration. Si on a un algorithme ZPP, on peut décider de répondre toujours 0, ou toujours 1, quand l'algorithme répond « je ne sais pas ». Cela démontre que $\mathsf{ZPP} \subseteq \mathsf{RP}$, et $\mathsf{ZPP} \subseteq \mathsf{coRP}$. Dans l'autre sens, on suppose disposer d'un algorithme RP et d'un algorithme coRP pour un même problème. Sur une entrée donnée, on exécute les deux algorithmes. S'ils répondent la même chose, on renvoie cette réponse qu'on sait correcte (car l'un ne fait pas de faux négatif, et l'autre pas de faux positif). Sinon, on répond « je ne sais pas ». La probabilité de répondre « je ne sais pas » est $\leq 1/3$. Donc $\mathsf{RP} \cap \mathsf{coRP} \subseteq \mathsf{ZPP}$.

Pour montrer que $\mathsf{RP} \subseteq \mathsf{NP}$, on peut formaliser un algorithme probabiliste comme étant un algorithme déterministe (standard) avec une entrée supplémentaire r (une suite de bits), qui est tirée au hasard. Au lieu d'appeler `random_bit`, il utilise les bits successifs de r au cours du calcul. Dire qu'un algorithme A est de type RP signifie que pour une entrée positive, une (large) majorité des valeurs possibles de r conduisent A à répondre OUI alors que pour une entrée négative, A renvoie toujours NON quelque soit r. Donc cet algorithme est bien un algorithme de vérification au sens de NP : la différence est que dans NP il suffit d'avoir *un* certificat valable sur les entrées positives (et on en a ici un grand nombre).

Pour l'inclusion $\mathsf{BPP} \subseteq \mathsf{PSPACE}$, la preuve est quasi-identique à la preuve $\mathsf{NP} \subseteq \mathsf{PSPACE}$: on simule tous les choix aléatoires possibles (toutes les valeurs de r dans la formalisation précédente). Ça prend un temps certes exponentiel, mais on peut le faire en espace polynomial. □

Exercice 3.6. Démontrer les inclusions restantes (qui sont évidentes) : $\mathsf{P} \subseteq \mathsf{ZPP}$ et $\mathsf{RP} \subseteq \mathsf{BPP}$.

R On peut imaginer résoudre un problème de NP par tirage aléatoire du certificat. Cela ne fonctionne pas car il peut n'exister qu'un seul certificat, donc la probabilité de succès peut être infime.

§ **Encore d'autres classes ?** Bien sûr, de nombreuses autres classes ont été définies et étudiées, correspondant au temps parallèle, à la complexité sur un ordinateur quantique, et plein d'autres mesures bien plus exotiques. Je renvoie à l'ouvrage de S. Perifel pour plus de détails. Une liste (non exhaustive mais assez complète) de 545 classes de complexités étudiées est disponible dans le *Complexity Zoo* : `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`.

RÉSUMÉ
- Des centaines de classes diverses peuvent être étudiées en théorie de la complexité.
- Les classes principales sont celles de complexité en temps et en espace, déterministes, non-déterministes, et probabilistes.