

Informatique

CPGE

Marc de Falco

Copyright © 2020 Marc de Falco

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Table des matières

I	Introduction	
1	Introduction	13
I	Feuille de route	13
2	Introduction à l'informatique	15
I	Données, Problème, Algorithme	15
I.1	Données	15
I.2	Problèmes	16
I.3	Algorithme	16
II	Modèle de calcul, Machines, Programmes	16
II.1	Exemple : ordinateur	17
II.2	Exemple : les machines à deux compteurs	17
II.3	Exemple : le noyau fonctionnel pur d'OCaml	18
II.4	Exemple : Jeu de la vie	18
II.5	Exemple : FRACTRAN	18
II.6	Modèle Turing complets	22
III	Langages, Compilateur, Interprète	22
III.1	Langages	22
III.2	Compilateur	23
III.3	Interprète	23
III.4	Unité de compilation, modules	23
IV	Paradigmes	23
IV.1	Impératif structuré	23
IV.2	Fonctionnel déclaratif	24
IV.3	Programmation logique	24

3	Introduction à la programmation (impérative)	27
I	Introduction	27
I.1	Les quatre étages de la programmation	28
I.2	Exemple	28
I.3	Se remettre sans cesse à l'ouvrage	29
II	Représenter	30
II.1	Les données simples	30
II.2	Les données composées ou structurées	30
III	Manipuler	30
III.1	Variables	30
III.2	État d'exécution	31
III.3	Instructions et blocs	31
III.4	Portée	32
III.5	Manipuler des données composées mutables	32
III.6	Instruction conditionnelle	34
III.7	Entrées et sorties	37
IV	Répéter	37
IV.1	Répéter n fois	37
IV.2	Répéter pour chaque élément	37
IV.3	Répéter tant qu'une condition est vérifiée	37
IV.4	Choisir la structure de boucle adaptée	37
IV.5	Boucles imbriquées et dimensionnalité	37
V	Abstraire	37
V.1	Définir et appeler des fonctions	37
V.2	Action d'une fonction et valeur de retour	37
V.3	Fonctions et portée	37
V.4	Structurer des programmes avec des fonctions	37
V.5	Les fonctions comme valeurs	37
4	Récursivité	39
I	La récursivité	39
I.1	Principe et exemples	39
I.2	Implémentation pratique	40
I.3	Programmer en récursif	41
I.4	Récursivité croisée	41
II	L'arbre d'appels	41
II.1	Définition	41
II.2	Complexité en nombre d'appels	41
II.3	Terminaison	41
III	Récursivité terminale	41
III.1	Présentation	41
III.2	Optimisation	41
III.3	Techniques	41
IV	Types inductifs et induction structurelle	41
IV.1	Définition naïve des types inductifs	41

5	Type option en OCaml	43
I	Principe	43
II	Syntaxe	43
III	Utilisation concrète	44
III.1	Définir une fonction partielle	44
III.2	Appeler une fonction partielle	45
III.3	Données partielles	45
III.4	Chaîne de traitement	45
6	Exceptions en OCaml	47
I	Syntaxe des exceptions	47
II	Exceptions pour la gestion d'erreurs	48
III	Programmer avec des exceptions	49
III.1	Retour prématuré	49

III

Structures de données

7	Structures de données abstraites et implémentations	55
I	Introduction	55
II	Structure de données abstraite	55
8	Séquences et ses implémentations : tableaux, listes chaînées	57
I	Structure abstraite séquence ou liste	57
II	Implémentations	57
III	Implémentations concrètes des Listes chaînées	57
III.1	En C	57
III.2	En OCaml	63
III.3	Structure de la mémoire	66
IV	Travaux pratiques	66
IV.1	Tableaux non statiques et tableaux dynamiques	66
IV.2	Listes chaînées	73
9	Piles et files : structures abstraites et implémentations	85
I	Piles	85
I.1	Principe	85
I.2	Représentation visuelle	86
I.3	Implémentations	86
I.4	Applications	88
II	Files	88
II.1	Principe	88
II.2	Implémentations	88
II.3	Applications	90

10	Arbres	91
I	Arbres binaires	91
I.1	Définition inductive	91
I.2	Vocabulaire	92
I.3	Implémentations	93
I.4	Arbres binaires stricts	94
I.5	Arbres binaires complets	95
II	Arbres	95
II.1	Définition	95
II.2	Implémentation	96
II.3	Représentation par un arbre binaire	97
II.4	Applications	98
III	Parcours	98
III.1	Parcours récursif d'un arbre binaire	98
III.2	Parcours impératifs	100
III.3	Parcours d'arbres	102
IV	Arbres binaires de recherche	102
IV.1	Objectif	102
IV.2	Définition	103
IV.3	Opérations	104
IV.4	Équilibrage	107
V	Tas	115
V.1	Présentation	115
V.2	Opérations	115
V.3	Implémentation	117
V.4	Application au tri	119
V.5	Application aux files de priorité	120
VI	TP	120
VI.1	Arbres en OCaml	120
VI.2	Arbres non binaires, tries	129
VI.3	Tries	132
VI.4	Arbres binaires en C, ABR, arbres rouges et noirs	136
11	Graphes	145
I	Graphes orientés	145
I.1	Définition	145
I.2	Voisins et degrés	146
I.3	Chemin	146
I.4	Sous-graphe	147
I.5	Implémentation	148
II	Graphes non orientés	149
II.1	Définition et adaptation du vocabulaire	149
II.2	Connexité	149
III	Graphe acyclique connexe	150
IV	Parcours	150
IV.1	Principe	150
IV.2	Profondeur	151

IV.3	Largeur	154
V	Chemin de plus petit poids	157
V.1	Graphe pondéré	157
V.2	Dijkstra : source unique	157
V.3	TODO Floyd-Warshall : toutes paires	161

IV

Algorithmique

12	Introduction à l'analyse des algorithmes	165
I	État d'un programme	165
II	Terminaison	166
II.1	Variant de boucle	167
II.2	Exemple de la recherche dichotomique	168
II.3	Exemple du tri à bulle	169
II.4	Lien avec la récursivité	171
II.5	Boucles imbriquées	171
III	Correction	171
III.1	Invariant de boucle	171
III.2	Exemple du tri par sélection	172
IV	Complexité	173
IV.1	Complexité dans le pire des cas	173
IV.2	Comparer des complexités	174
IV.3	Complexités en temps classiques	177
IV.4	Calculer des complexités	180
IV.5	Complexité à plusieurs paramètres	180
IV.6	Complexité en moyenne	181
IV.7	Complexité amortie	182
IV.8	Pertinence de la complexité spatiale	182
V	Exercices	183
13	Complexité amortie	187
I	Introduction	187
I.1	Implémentation d'une file avec deux piles	187
I.2	Les tableaux dynamiques	188
II	Techniques de calcul	191
II.1	Cadre	191
II.2	Calcul naïf	191
II.3	Méthode du banquier	192
II.4	Méthode du potentiel	195
14	Recherche par Force brute	197
I	Principe	197
I.1	Problème de décision et exploration exhaustive	197
I.2	Problème d'optimisation et exploration exhaustive	198

II	Recherche par retour sur trace (backtracking)	198
II.1	Construction itérative de candidats	198
II.2	Évaluation partielle et raccourcis	200
II.3	Énumération de toutes les solutions	203
II.4	TP : tours du cavalier	205
II.5	TP : jeu du solitaire	205
III	Stratégies d'énumération	210
III.1	Combinatoire élémentaire	210
III.2	Enumération d'arbres	211
IV	Droite de balayage	211
IV.1	Principe	211
IV.2	Plus proche paire	211
15	Algorithmes gloutons	219
I	Principe	219
I.1	Problème d'optimisation combinatoire	219
I.2	Algorithme glouton	220
I.3	Cas du rendu de monnaie	221
II	Construction de l'arbre de Huffman	222
II.1	Description	222
II.2	Algorithme glouton et implémentation	223
II.3	Preuve d'optimalité	224
III	Sélection d'activités	225
III.1	Description	225
III.2	Algorithme glouton et implémentation	226
III.3	Preuve d'optimalité	227
IV	Principe général des preuves d'optimalité	228
V	Ordonnancement de tâches	229
V.1	Description	229
V.2	Algorithme glouton et implémentation	230
V.3	Preuve d'optimalité	232
VI	Exercices	233
16	Division en sous-problèmes	237
I	Diviser pour régner	237
I.1	Principe	237
I.2	Somme d'éléments dans un tableau	238
I.3	Tri fusion	239
I.4	Recherche dichotomique	241
I.5	Principe général d'analyse des récurrences	241
I.6	Nombre d'inversions	243
I.7	Points les plus proches	245
II	Meet in the middle	246
II.1	Principe	246
II.2	Sous-ensemble de somme donnée	246

III	Dichotomie pour passer de décision à optimisation	246
III.1	Principe	246
III.2	Couverture par des segments égaux	246
IV	Problèmes	248
IV.1	Multiplication d'entiers	248
17	Algorithmique des textes	251
I	Recherche dans un texte	251
I.1	Principe de la recherche	251
I.2	Algorithme naïf en force brute	252
I.3	Algorithme de Boyer-Moore	253
I.4	Algorithme de Rabin-Karp	264
II	Compression	267
II.1	Principe	267
II.2	Algorithme d'Huffman	268
II.3	Algorithme de Lempel-Ziv-Welch	270
III	Problèmes supplémentaires	278
III.1	Transformation de Burrows-Wheeler	278
III.2	Move to front	278
III.3	La structure de données corde	278
III.4	L'algorithme de Knuth-Morris-Pratt	278
III.5	Extensions à l'analyse d'images	278

V

Systèmes

18	Gestion de la mémoire dans un programme compilé	281
I	Organisation de la mémoire	281
II	Pointeurs	284
III	Portée d'un identificateur	284
IV	Piles d'exécution, variables locales et paramètres	286
V	Allocation dynamique	288
V.1	Allocation	288
V.2	Libération	289
V.3	Protection mémoire	290
V.4	Réalisation d'un système d'allocation de mémoire	291



Introduction

1	Introduction	13
I	Feuille de route	
2	Introduction à l'informatique	15
I	Données, Problème, Algorithme	
II	Modèle de calcul, Machines, Programmes	
III	Langages, Compilateur, Interprète	
IV	Paradigmes	

1. Introduction

Ce site présente mon poly de cours autour du programme de MP2I/MPI, et donc également du programme d'option informatique de MPSI/MP et du tronc commun.

Il est actuellement en cours d'écriture.

I Feuille de route

- Faire une présentation uniforme pour les problèmes avec un index.
- Rajouter des macros pour compiler/exécuter et inclure la sortie des bouts de code.
- Rajouter les TP.
- Des roadmaps sont données dans chaque chapitre.



2. Introduction à l'informatique

■ Note 2.1 Roadmap :

- tout reprendre. Je suis peu satisfait du résultat.

Dans ce chapitre, on présente de manière très survolée l'informatique. On aura l'occasion de revenir en détail sur certaines notions. L'objectif est avant tout d'avoir une vision claire même si elle est naïve de

- ce qu'est un problème informatique
- ce qu'est un programme
- ce qu'est une machine
- ce que signifie de résoudre informatiquement un problème

Cela permet de répondre à des questions comme :

- qu'est-ce qui permet de dire qu'on programme en OCaml alors qu'on a l'impression de ne faire que des suites de définitions ?
- peut-on se contenter de programmer sans chercher à comprendre ?

Précisons ici qu'on parle d'informatique mais que le terme le plus proche de ce que l'on étudie en anglais est *computer science* par complémentarité avec des notions comme celles de *computer engineering* plus proches de considérations matérielles. Il n'y a pas une *unique* vision de l'informatique mais plutôt une grande richesse de point de vue.

I Données, Problème, Algorithme

I.1 Données

Définition I.1 Une donnée informatique est un élément d'information fini qu'on peut stocker ou transmettre.

Selon le niveau d'abstraction auquel on se place, on peut considérer une donnée comme étant une succession de bits, c'est-à-dire de valeur valant 0 ou 1, ou comme une donnée plus structurées, comme un texte, une image, un graphe, ...

En informatique, on adapte souvent son point de vue sur les notions de base comme celle-ci. Si

on fait de la transmission ou de la compression de données, c'est important d'en avoir une représentation la plus primitive possible comme des mots binaires. Mais si s'intéresse à des notions de chemins dans un graphe, il est plus adéquat de considérer un graphe comme un élément de données même si la question de sa représentation n'est pas immédiate.

1.2 Problèmes

Définition 1.2 Un *problème informatique* est un ensemble de questions paramétrées par des données (les *entrées*) et dont les réponses peuvent être

- soit Oui ou Non, auquel cas on parle de problème de *décision*
- ou d'autres données dépendant des entrées : les *sorties*, on parle alors de problème de *recherche* ou de *construction*.

■ **Exemple 2.1** • Étant donné un entier, déterminer s'il est premier ou non.

- Étant donnés deux entiers, calculer leur PGCD.
- Étant donné un graphe pondéré et deux sommets, déterminer un chemin de longueur minimale permettant d'aller de l'un à l'autre.
- Étant un programme informatique et une entrée, déterminer si le programme va s'arrêter ou non.
- Étant donné une image et une base de donnée de visages, déterminer les personnes présentes sur l'image.

On voit que certains de ces problèmes n'ont pas eu besoin d'attendre l'informatique, comme les problèmes d'arithmétique.

On peut prouver qu'un problème comme le problème de l'arrêt ne pourra jamais être résolu par un ordinateur (on dira qu'il est *indécidable*).

1.3 Algorithme

Définition 1.3 Un *algorithme* est un procédé systématique et mécanisable permettant de résoudre un problème informatique.

Ce qu'on signifie ici par le mot *systématique* est que le procédé de calcul est précis et non ambigu et par *mécanisable* qu'il s'agit d'un calcul qui puisse être fait par une *machine* dans un sens à venir.

Une autre manière de voir les problèmes est de dire que ce sont des fonctions mathématiques des entrées vers les sorties. Un algorithme est alors une description de la réalisation d'une telle fonction.

II Modèle de calcul, Machines, Programmes

On donne ici des définitions volontairement naïves et imprécises. Elles pourraient être précisées en fixant un cadre théorique plus conséquent, ce qui est prématûr à ce stade.

Définition II.1 Un modèle de calcul est un cadre permettant de décrire comment calculer la solution de problèmes en fonction des entrées.

Définition II.2 Une *machine* est un modèle de calcul défini par un ensemble de *configurations* et des opérations élémentaires, aussi appelées *instructions*, permettant de passer d'une configuration à une autre.

Ces instructions sont regroupées sous la forme d'une suite finie appelée un *programme*.

Pour exécuter un programme, il y a dans la configuration un entier qui s'appelle le *pointeur*

d'instruction et qui donne le numéro dans le programme de l'insrtuction courante. Généralement, on part de la première instruction en passant ensuite, sauf contre-ordre, à l'instruction suivante, jusqu'à la fin du programme.

II.1 Exemple : ordinateur

Un ordinateur est naturellement un exemple de machine. Une configuration est la donnée de l'état des registres du processeur, de la mémoire et aussi des périphériques d'entrée-sortie. Un programme est une suite d'instructions exécutées par le processeur.

II.2 Exemple : les machines à deux compteurs

Les configurations d'une machine à deux compteurs sont des triplets (A, B, PC) où A et B sont deux compteurs entiers naturels, en général initialisés avec les entrées et dans lesquels on pourra lire les sorties, et PC est le pointeur d'instruction, initialisé à la première instruction.

Un programme est une suite finie et numérotée d'instructions élémentaires parmi les suivantes :

- INCA : incrémenter A , c'est-à-dire ajouter 1 à la valeur qu'il contient
- DECA : décrementer A , c'est-à-dire soustraire 1 à la valeur qu'il contient
- IFA $i \ j$ sauter à l'instruction numéro i si A est nul, sinon sauter à l'instruction numéro j .
- les instructions INCB, DECB et IFB $i \ j$ respectives vis-à-vis du compteur B .

Ainsi, on peut considérer le programme suivant

```

1: IFA 6 2
2: DECA
3: INCB
4: INCB
5: IFA 1 1
6: INCA
7: DECA

```

Si on initialise les compteurs (A, B) à $(n, 0)$ et qu'on exécute le programme, on obtient alors $(0, 2n)$ dans les compteurs. Ce programme réalise ainsi un doublement du compteur A et place le résultat dans le compteur B .

Exercice 2.1 1. Écrire un programme permettant de passer de la configuration (a, b) à $(0, a + b)$.
 2. Écrire un programme permettant de passer de la configuration $(n, 0)$ à $(0, 0)$ si n est pair ou $(0, 1)$ sinon.

1.

```

1: IFA 5 2
2: DECA
3: INCB
4: IFA 1 1
5: INCA
6: DECA

```

2.

```

1: IFA 7 2
2: DECA
3: IFA 6 4
4: DECA
5: IFA 1 1
6: INCA
7: INCA
8: DECA

```

II.3 Exemple : le noyau fonctionnel pur d'OCaml

On considère ici le noyau OCaml avec les entiers, leurs opérations élémentaires, les fonctions, les définitions (récursives ou non) et ce qui correspond à l'évaluation.

Ainsi le terme OCaml suivant `fun n -> 2 * n` peut être vu simplement comme prenant en entrée un entier n et renvoyant son double.

Ce qui est important ici, sans qu'on s'attarde sur la définition précise de l'évaluation, c'est de comprendre qu'il s'agit d'un modèle de calcul sans avoir de notion de machine, d'instructions ou de programme.

Bien entendu, quand on interprète ou qu'on compile un programme OCaml, cela se passe sur un ordinateur, donc une machine.

On considère que les termes sont les programmes dans un tel langage de programmation. La différence principale entre un programme vu ainsi et un dans le sens usuel, comme un programme Python, c'est que l'on n'a pas conscience de la réalité la machine quand on programme : il n'y a pas de notion d'instructions. Bien sûr, on verra qu'il est possible de retrouver cela dans OCaml et ainsi de se ramener en terrain connu, mais c'est important de comprendre que ce noyau existe indépendamment d'une notion de machine.

■ Remarque 2.1 C'est Alonzo Church qui a défini ce noyau, qu'on appelle le λ -calcul, en 1933 alors qu'il souhaitait présenter un cadre pour exprimer les fonctions calculables. ■

II.4 Exemple : Jeu de la vie

Le jeu de la vie défini en 1970 par John Conway dans le but d'être une récréation mathématique s'est révélé être un modèle de calcul très important.

Il s'agit d'un cas particulier d'une plus grande classe de modèles de calcul appelés les **automates cellulaires**.

Une configuration du jeu de la vie est une grille bidimensionnelle de valeur booléenne. On appelle chaque case une *cellule* et on dirait que le booléen permet de déterminer si elle *vivante* ou *morte*.

L'évaluation de ce modèle consiste à définir une nouvelle configuration en appliquant la règle suivante pour chaque cellule :

- si la cellule est vivante et a deux ou trois voisines vivantes, elle reste vivante dans la nouvelle configuration, sinon elle meurt.
- si la cellule est morte, elle devient vivante si et seulement si elle a exactement trois voisines vivantes.

Quand on parle de cellules voisines, on fait référence aux huit voisines directes sur la grille (on compte ainsi les diagonales).

Le point essentiel dans cette procédure permettant de passer d'une configuration à une autre est qu'on ne modifie pas la configuration courante, tout se passe comme si chaque cellule était modifiée simultanément.

La configuration initiale est à la fois le programme et ses entrées. Un point critique ici c'est qu'il n'existe pas de notion de fin d'évaluation. En effet, même des configurations très simples n'atteignent pas un point fixe. On devra donc choisir en fonction du problème que l'on cherche à résoudre, comment déterminer la fin du calcul.

II.5 Exemple : FRACTRAN

FRACTRAN est un modèle de calcul également inventé par John Conway en 1987.

Un programme FRACTRAN est une suite finie de fractions d'entiers naturels comme :

$$P = \left(\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{2}, \frac{1}{7}, \frac{55}{1} \right)$$

Une entrée est un entier et pour exécuter un tel programme, on considère un entier courant n , initialisé à la valeur de l'entrée, et on parcourt les fractions de gauche à droite jusqu'à trouver une fraction $\frac{p}{q}$ tel que $n \frac{p}{q}$ soit un entier. Dans ce cas, c'est la nouvelle valeur de n et on recommence le processus. Sinon, le programme termine.

■ **Exemple 2.2** ($\frac{3}{10}, \frac{4}{3}$) sur l'entrée 14 va s'arrêter tout de suite, sur l'entrée 15, on va avoir 20, 6 puis 8 et s'arrêter. ■

Exercice 2.2 Exécuter P sur l'entrée 2 suffisamment longtemps pour noter les quatre premières puissances de 2 prises par l'entier courant. Attention, cela nécessite sûrement de programmer l'évaluation. ■

Démonstration.

On remarque que les puissances de 2 qui apparaissent sont dans l'ordre : $2^2, 2^3, 2^5, 2^7, 2^{11}, 2^{13}, \dots$. Ainsi ce programme énumère les nombres premiers. ■

Comment fonctionne FRACTRAN ? En fait, chaque fraction va être de la forme

$$\frac{p_1^{a_1} \cdots p_k^{a_k}}{q_1^{b_1} \cdots q_l^{b_l}}$$

où les p_i et les q_j sont des nombres premiers distincts.

Pour qu'on sélectionne cette fraction, il faut que n soit divisible par le dénominateur, donc si on écrit n en produit de facteurs premiers, il faut que les puissances correspondant à chaque q_i soit $\geq b_i$. On va alors les diminuer de b_i puis augmenter les puissances correspondant à chaque p_i de a_i .

Ainsi, si on considère qu'un nombre n s'écrit $n = \prod_{i \in \mathbb{N}} p_i^{a_i}$, avec les a_i presque tous nuls et $p_0 < p_1 < p_2 < \dots$ une énumération des nombres premiers, on voit qu'un nombre n'est alors qu'une configuration d'une infinité de compteurs indexés par des entiers et qui sont presque tous à 0. Une fraction est alors une règle de la forme :

$$C[i_1] \geq b_1, \dots, C[i_k] \geq b_k \rightarrow C[i_1] - b_1, \dots, C[i_k] - b_k, C[j_1] + a_1, \dots, C[j_l] + a_l$$

où on a nommé $C[0], C[1], \dots$ les compteurs.

FRACTRAN est donc une sorte de machine à compteurs dont les instructions sont exécutées comme dans un filtrage OCaml en cherchant de gauche à droite la première règle qui s'applique. Le point le plus délicat c'est que la condition permettant d'appliquer une règle est destructrice : on divise, et on ne peut pas utiliser le même compteur pour la condition et pour le résultat, car sinon on simplifierait la fraction.

■ **Remarque 2.2 Comment fonctionne le programme P ?**

Pour cela, on va réécrire P en suivant la syntaxe précédente, comme on a forcément des décrements de compteurs correspondant à la condition d'application, il est inutile de les répéter et on écrira juste

2:-2 3:-2 |> 1:+2 0:+1

pour la fraction $\frac{18}{1225} = \frac{2 \cdot 3^2}{5^2 \cdot 7^2}$ car $p_0 = 2, p_1 = 3, p_2 = 5$ et $p_3 = 7$. Par soucis de lisibilité, on peut de plus nommer les compteurs par des lettres en commençant à a et donc écrire

c-2 d-2 |> b+2 a+1

Comme les règles de P ne font intervenir que des $+1$ et des -1 , on pourra encore abréger en écrivant $a-$ plutôt que $a-1$ et $b+$ plutôt que $b+1$.

Ainsi, P devient la suite de règles suivantes (numérotées pour y faire référence ensuite).

0:	d-f-	>	g+
1:	c-g-	>	a+b+f+
2:	b-g-	>	h+
3:	a-h-	>	i+
4:	b-e-	>	j+
5:	j-	>	d+e+
6:	i-	>	c+h+
7:	h-	>	d+e+
8:	g-	>	
9:	f-	>	e+
10:	e-	>	f+
11:	a-	>	b+c+
12:	d-	>	
13:		>	c+e+

On commence avec la configuration $n = 2 = 2^1$ donc uniquement 1 dans le compteur a et 0 dans les autres. On notera

$a : 1$

cette configuration.

On remarque donc que si on a la configuration

$a : n - 1$

on va appliquer la règle 11 jusqu'à obtenir la configuration

$b : n - 1, c : n - 1$

Ensuite, on applique la règle 13 puis successivement les règles 4 et 5 jusqu'à être dans la configuration

$c : n, d : n - 1, e : 1$

La règle 10 permet de passer à

$c : n, d : n - 1, f : 1$

On va effectuer une succession de règle afin de déterminer si $n - 1$ divise n . En fait, cette succession de règle va déterminer si d divise c . Imaginons donc qu'on a la configuration

$$c : n, d : m, f : 1$$

on applique successivement les règles 0 et 1 pour obtenir la configuration

$$a : m, b : m, c : n - m, f : 1$$

La règle 9 s'applique alors pour passer à

$$a : m, b : m, c : n - m, e : 1$$

et on effectue alors la succession des règles 4 et 5 qu'on a déjà vu pour passer à

$$a : m, c : n - m, d : m, e : 1$$

Comme il n'y a plus rien dans b c'est la règle 10 qui s'applique et on repasse à

$$a : m, c : n - m, d : m, f : 1$$

On repasse alors dans les règles 0 et 1 jusqu'à obtenir

$$a : 2m, b : m, c : n - 2m, f : 1$$

et ainsi de suite. Pour l'arrêt, deux cas peuvent alors se produire :

- Soit on a vidé c en effectuant la soustraction de m , ce qui correspond au cas où m ne divise pas n , et alors la règle 1 ne pourra plus s'appliquer car elle nécessite $c \geq 1$. Si $n = qm + r$ avec $0 < r < n$, on aboutit alors avec une dernière application de 0 en

$$a : n, b : r, d : m - r - 1, g : 1$$

Comme la règle 1 ne s'applique plus, on passe alors à la règle 2 vers

$$a : n, b : r, d : m - r - 1, h : 1$$

on enchaîne alors les règles 3 et 6 jusqu'à obtenir la configuration

$$b : r - 1, c : n, d : m - r - 1, h : 1$$

On ne peut alors qu'apppliquer la règle 7 qui permet de passer à

$$b : r - 1, c : n, d : m - r, e : 1$$

puis l'alternance 4 et 5 transfère b dans d vers

$$c : n, d : m - 1, e : 1$$

Dans ce cas, si on a $m - 1 > 0$ et on va alors repasser dans la règle 10 pour tester la divisibilité de n par $m - 1$. Mais ce qui est subtil ici, c'est que le cas $m - 1 = 0$ a déjà été pris en compte car il correspond à $m = 1$, c'est-à-dire à une divisibilité qui aboutit forcément et qui est donc traité par le cas suivant.

- Soit m divise n , donc $n = km$ et on va aboutir forcément par soustraction après une règle 0 à une configuration de la forme

$$a : n, d : m - 1, g : 1$$

Donc, contrairement au cas précédent, la règle 2 ne s'applique pas car $b = 0$. On passe donc à la règle 8 qui produit

$$a : n, d : m - 1$$

C'est ici qu'on peut produire un nombre premier, car si $m - 1 = 0$, cela veut dire que la seule divisibilité s'est produite pour 1, donc que n est premier. Or, on a alors uniquement n dans le compteur a , et ainsi ça correspond au 2^n .

On poursuit alors par une étape de nettoyage pour passer à l'entier $n + 1$ et continuer. Pour cela, on applique la règle 11 comme précédemment. Le seul changement c'est qu'avant de passer à la règle 13, on va appliquer la règle 12 pour vider le compteur d .

Ainsi, ce programme pourrait être traduit par le code Python suivant :

```

Python
a = 1
while True:
    a = a+1
    for d in reversed(range(1,a)):
        b = a
        reste = False
        while b != 0:
            for i in range(d):
                if b == 0:
                    reste = True
                    break
                b = b-1
            if not reste:
                break
        if d == 1:
            print(a, 'est premier')
```

II.6 Modèle Turing complets

Tous les exemples présentés ci-dessus sont équivalents : ils permettent de calculer les mêmes fonctions mathématiques pourvu qu'on précise la manière dont on donne l'entrée et comment on lit la sortie.

Par exemple, s'il n'est pas possible de réaliser l'incrémentation en FRACTRAN, il est possible d'écrire un programme permettant de passer de 2^a à 3^{a+1} .

On dit que ces modèles sont **Turing-complet** en référence au modèle de calcul de référence : les machines de Turing.

III Langages, Compilateur, Interprète

III.1 Langages

Un langage de programmation est une manière de représenter textuellement un programme. Cela peut être un programme en lien avec une notion de machine mais aussi une notion plus abstraite comme on va le voir.

Comme on l'a vu, un ordinateur est une machine. Un programme pour cette machine est ce qu'on appelle un binaire et l'ensemble des instructions s'appelle le langage machine. C'est le langage naturel d'un ordinateur et il dépend de son processeur. Ainsi, un binaire pour un téléphone n'est pas directement utilisable sur son ordinateur, car les premiers ont des processeurs utilisant un jeu d'instructions arm et les seconds un jeu d'instructions x86_64.

III.2 Compilateur

Un **compilateur** est un traducteur d'un langage à un autre. On considère le plus souvent des compilateurs vers un langage machine, qu'il soit lié à des processeurs réels ou une machine spécifique appelée une machine virtuelle.

III.3 Interprète

Un interprète est un programme qui va lire un programme écrit dans un langage donné et l'évaluer en interprétant les instructions. Cela diffère d'une machine virtuelle dans le fait que le programme utilise toute la richesse du langage de programmation dans lequel il a été programmé pour son interprète. De plus, le fait de ne pas passer par une machine virtuelle permet de plus facilement relier l'environnement de l'interprète et celui du langage interprété. Un autre avantage discutable et la possibilité dans le langage cible de faire référence à l'interprète. Cela permet notamment de permettre l'évaluation dynamique de code (`eval` en Python).

Pour autant, les interprètes ont de nombreux défauts qui font qu'on a de plus en plus recours à des compilateurs vers des machines virtuelles. Parmi ceux-ci, citons le fait de devoir relire directement le programme à chaque fois qu'on va l'exécuter, ce qui est particulièrement critique quand on veut faire appels à des fonctions définies ailleurs, ou le fait que l'interprète lui-même est souvent très verbeux.

III.4 Unité de compilation, modules

Un programme est souvent structuré sous la forme de nombreux sous-programmes reliés entre eux. C'est une bonne pratique pour regrouper des fonctions selon un même thème et permettre la réutilisation du code sans duplication. On utilise pour cela la notion de modules ou de bibliothèques. Cela correspond au niveau du programme à rajouter du code existant au moment de la compilation ou de l'interprétation. Il est alors possible de réutiliser une partie de la compilation sur ce code, et ainsi, on utilise en général une version intermédiaire qui est du code pré-compilé utilisable par un autre programme, soit en copiant le résultat dans le programme ou en faisant référence à des versions définies dans le système (bibliothèque dynamique). Un programme est chargé de faire le lien entre les différentes bibliothèques et le programme, c'est l'éditeur de liens.

IV Paradigmes

Un paradigme de programmation est une manière de concevoir des programmes. On va citer ici trois paradigmes.

IV.1 Impératif structuré

Un programme impératif est un programme construit plus ou moins directement en lien avec une notion de machines comme une suite d'instructions. Afin de permettre de programmer convenablement, un tel langage fournit une notion de structure qui permet de structurer le code. D'une certaine manière, la notion de classe et la programmation objet enrichit ce paradigme en permettant de structurer le code autour d'une notion d'objets.

IV.2 Fonctionnel déclaratif

On a vu qu'en programmation fonctionnelle, tout n'est qu'une expression, un terme, qu'on évalue. Un programme fonctionnel déclaratif est ainsi consistué d'une succession de définitions de valeurs ou de fonctions. C'est ainsi qu'en OCaml n'est qu'une suite de `let` ou `let rec`.

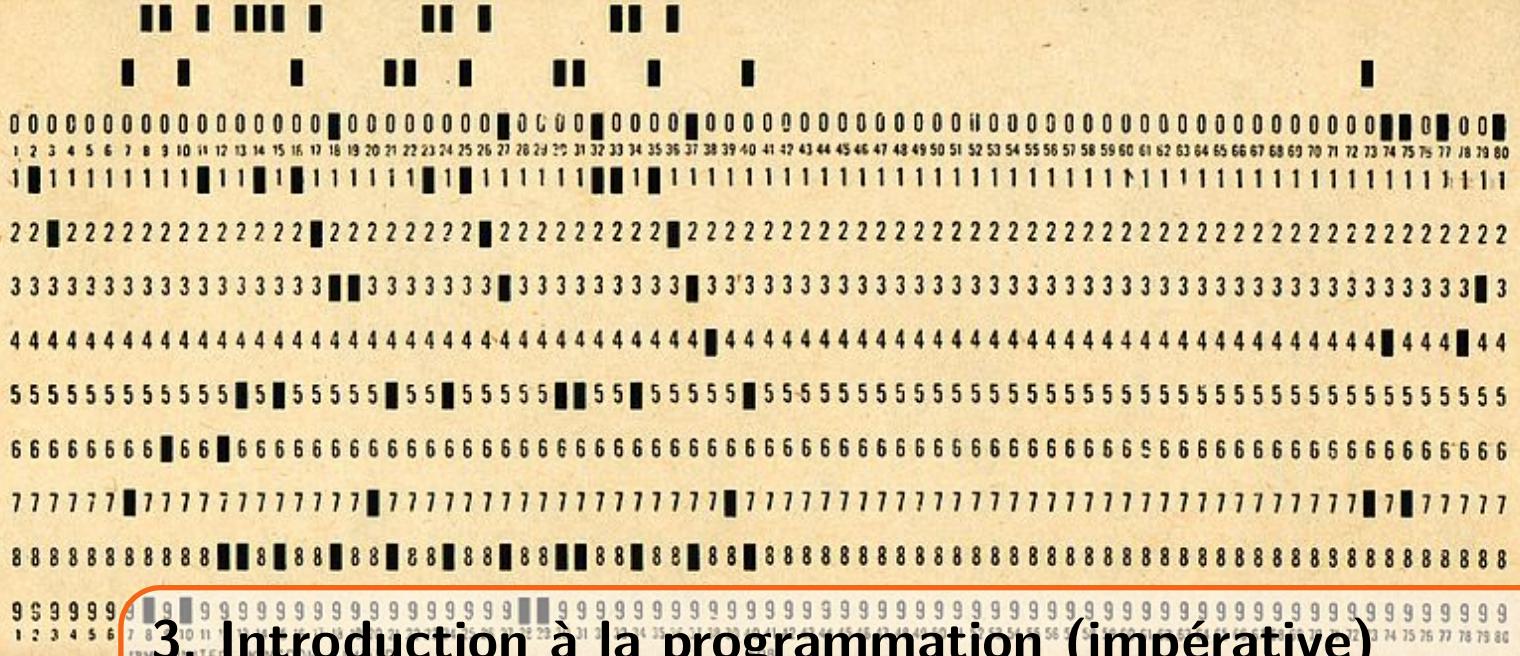
Comment calcule un tel programme ? En évaluant chacune des déclarations, certaines vont effectivement produire des effets de bords et permettre la lecture ou l'écriture. L'usage en OCaml est ainsi de finir un programme par une déclaration nommée `main` ou `_` qui va se charger d'appeler les déclarations précédentes.

IV.3 Programmation logique

Le paradigme de la programmation logique est sûrement le plus déroutant des trois. Un programme logique est en fait une formule logique dont une solution correspond à son résultat. Sans rentrer dans les détails, on remarquera que cela correspond à ce qu'on fait en SQL pour les bases de données où une requête de recherche est définie sous une forme de formule logique qui décrit les propriétés qui doivent être vérifiées par les éléments qu'on cherche.

Programmation

3	Introduction à la programmation (impérative)	27
I	Introduction	
II	Représenter	
III	Manipuler	
IV	Répéter	
V	Abstraire	
4	Récursivité	39
I	La récursivité	
II	L'arbre d'appels	
III	Récursivité terminale	
IV	Types inductifs et induction structurelle	
5	Type option en OCaml	43
I	Principe	
II	Syntaxe	
III	Utilisation concrète	
6	Exceptions en OCaml	47
I	Syntaxe des exceptions	
II	Exceptions pour la gestion d'erreurs	
III	Programmer avec des exceptions	



3. Introduction à la programmation (impérative)

Source de l'image : <https://www.flickr.com/photos/binaryape/5151286161/>

■ Note 3.1 Roadmap :

- finir l'écriture avec Python.
- rajouter OCaml et C.
- en profiter pour faire une introduction aux référence en OCaml quitte à rajouter une partie spécifique.
- rajouter beaucoup d'exemples et d'exercices.

■ Introduction

Dans ce chapitre introductif sur la programmation, on va présenter celle-ci à travers quatre notions permettant de comprendre ce que signifie programmer. Un certain soin a été porté au fait de rendre cette présentation indépendante du langage tout en permettant qu'elle serve de support pour l'apprentissage d'un langage particulier. Ainsi, les éléments propres à un langage donnée sont clairement séparé.

Comme nous l'avons vu dans le chapitre d'introduction, la programmation consiste à décrire des algorithmes et la manière dont ils sont mis en œuvre sur une machine. En cela, le modèle que l'on suit ici est celui de la programmation dite impérative structurée. Pour autant, de nombreuses notions présentées ici restent valides dans d'autres paradigmes.

La présentation faite ici est indépendante du langage sous réserve que celui-ci comporte des traits impératifs structurés. Pour autant, selon les langages certaines notions sont plus ou moins riches. Des paragraphes spécifiques complètent ainsi le texte et on encourage le lecteur à basculer d'un langage à un autre tout au long de la lecture.

C est un langage ayant peu d'expressivité directe pour les données. On est donc très vite réduit à devoir définir ses propres structures pour l'enrichir. Or, c'est plus complexe que la mise en œuvre des notions impératives développées ici. Le lecteur complétera alors sa

lecture par le chapitre d'introduction aux structures de données en C.

1.1 Les quatre étages de la programmation

Programmer c'est

- **représenter** des données, allant du booléen valant vrai ou faux, jusqu'aux bases de données permettant de modéliser des relations complexes
- **manipuler** ces données en lien avec un environnement, ce qui signifie autant de pouvoir les stocker que de pouvoir les transformer
- **répéter** ces manipulations élémentaires, car les données sont d'une taille finie, mais non connue à l'avance
- **abstraire** les notions précédentes pour ne pas avoir à se répéter, à travers la notion de fonctions

1.2 Exemple

Prenons un exemple simple pour comprendre les différents étages où le but est final est de pouvoir calculer des sommes comme $1 + 2 + \dots + n$.

Représenter Un entier comme 3 est une donnée qu'on pourra représenter telle quelle dans la plupart des langages avec cependant des limitations à garder en tête : la mémoire étant finie, on ne peut pas représenter des entiers quelconques, car il y en a une infinité. Déjà, à ce stade, on peut se poser la question de savoir si on ne veut représenter que des entiers dans un intervalle fixe, afin de borner leur occupation mémoire, ou si on veut les représenter en arbitraire. On peut opérer sur ces entiers avec un opérateur tel que `+` pour l'addition, mais le choix de la représentation aura son importance sur ce qui se cache derrière cette opération. L'addition de deux très grands entiers en précision arbitraire prendra plus de temps.

A l'aide de cela, on peut ainsi écrire $1 + 2 + 3 + 4 + 5$.

Manipuler On peut aussi les stocker directement en mémoire à l'aide de variable. Ainsi

```
int x = 3;
```

permettra de définir une variable nommée `x` et contenant la valeur entière 3. On peut alors manipuler cette variable en changeant sa valeur à l'aide des opérations :

```
x = x + 2;
```

Avec ces instructions de manipulation, on peut reprendre le calcul précédent de $1 + 2 + \dots + 5$ avec une suite d'instructions :

```
int s = 0;
s = s + 1*1;
s = s + 2*2;
s = s + 3*3;
s = s + 4*4;
s = s + 5*5;
```

Répéter En fait, les 5 instructions d'ajout qu'on vient d'écrire ont la même structure : il s'agit d'ajouter une valeur i^2 à la variable `s` avec `i` prenant tour à tour les valeurs 1, 2, 3, 4 et 5. Une structure comme une boucle va permettre d'éviter cette répétition et d'écrire qu'il existe une variable `i` prenant ces valeurs et les instructions à réaliser pour chaque valeur de `i`. On pourra donc écrire :

```
C int s = 0;
for(int i = 1; i <= 5; i++)
{
    s = s + i*i;
}
```

C'est en fait très proche de la notation $1 + 2 + \dots + 5 = \sum_{i=1}^5 i$. En fait, on cette notation mathématique est très proche d'une boucle.

Abstraire imaginons qu'on ait besoin d'effectuer le calcul précédent $\sum_{i=1}^n i^2$ à plusieurs reprises pour différentes valeurs de n , on pourrait à chaque fois recopier ces lignes en changeant la valeur maximale prise par i . Mais c'est inefficace pour plusieurs raisons :

- en recopiant du code, on risque de propager et répéter des erreurs. Si jamais on découvre une meilleure manière de faire le calcul, il faudra ainsi changer le code à chaque endroit où on l'a copié.
- là où on a besoin de ce calcul, il est possible que ce soit pour faire d'autres choses avec et donc qu'il s'inscrive dans une logique complexe. En copiant le code, on rend le programme plus compliqué à comprendre, car la boucle pour calculer cette somme est mise sur le même plan que le code qui nous intéresse.

Pour faire un parallèle avec les mathématiques, c'est comme si on recopiait une preuve dans un cas particulier chaque fois qu'on a besoin d'appliquer un théorème.

On introduit ainsi une notion d'abstraction *les fonctions* qui vont nous permettre de faire un code générique de calcul en prenant n en paramètre.

```
C int somme_carres(int n)
{
    int s = 0;
    for(int i = 1; i <= n; i++)
    {
        s = s + i*i;
    }
    return s;
}
```

Comme on le verra, une telle fonction est caractérisée par son nom qui nous permet d'y faire référence, comme lorsqu'on applique le théorème de Thalès, ainsi que des arguments qui seront, à l'exécution, remplacé par les valeurs qui nous intéressent, comme le fait que Thalès est démontré dans un triangle *générique* mais on l'applique sur un triangle particulier et une valeur de retour qui correspond à ce que la fonction calcule.

On peut alors, quand on a besoin de faire le calcul de $\sum_{i=1}^5 i^2$ juste écrire :

```
C somme_carres(5)
```

I.3 Se remettre sans cesse à l'ouvrage

De notre présentation *étagée*, on pourrait retenir une hiérarchisation des différents étages. Il n'en est rien.

Ainsi, autant il peut être assez direct de représenter des données, comme dans le cas précédent, autant cela peut être une étape cruciale que de choisir la meilleure représentation. C'est le cas quand on s'intéresse au choix d'une structure de données la plus adaptée ou quand on conçoit un schéma de base de données.

La programmation, et plus largement l'informatique, sont ainsi à rapprocher d'un art martial comme l'Aïkido où on travaille tout au long de sa pratique les mêmes gestes simples en se

perfectionnant sans cesse. Aller au bout de ce chapitre ne sera donc pas le signe qu'on maîtrise la programmation, mais juste qu'on a fait le premier pas sur la voie du perfectionnement.

Tout au long des autres chapitres, on verra ainsi des méthodes, des notions, qui permettent de mieux comprendre et de mieux pratiquer la programmation. Mais au moment où se retrouvera à programmer, on ne sortira pas de ces quatre étages.

II Représenter

Les données sont regroupées en informatique autour de la notion de types. Un type de données peut être vu en première approximation comme un ensemble de données de même nature. On considère, par exemple, usuellement le type des entiers naturels ou celui des nombres à virgule flottante.

II.1 Les données simples

Les données les plus simples sont celles qui correspondent à des valeurs numériques. Elles dépendent assez souvent des langages de programmation, mais on retrouve toujours un type pour les booléens, un type pour les entiers et un type pour les nombres à virgule flottante, c'est-à-dire pour des représentations de certains nombres réels.



II.2 Les données composées ou structurées

Les données simples permettent de tout représenter. En effet, l'élément de donnée le plus primitif est le bit qui correspond à un booléen. Ainsi, la mémoire d'un ordinateur est entièrement constituée de booléens. Mais en disant cela, on ne dit pas grand-chose, car il ne s'agit pas d'une soupe informe de booléens, mais d'une organisation structurée. C'est ainsi qu'on considère des données composées comme les tableaux ou les couples.

On distingue deux types de données composées :

- les données *immuables*, c'est-à-dire celles qui ne permettent pas de changer les valeurs qu'elles regroupent ni leur structure après leur création
- les données *mutables* qui le permettent.



III Manipuler

III.1 Variables

L'élément clé permettant de manipuler des valeurs est de pouvoir les placer à un endroit et de changer la valeur qui s'y trouve. C'est ce qu'on appelle une **variable**. Même si cela ne correspond pas à une réalité selon les langages, il est d'usage de considérer une variable comme une case mémoire disposant d'un nom et dans laquelle on place des valeurs.

On a alors trois éléments :

- définir une variable, souvent en la limitant à un certain type de donnée, on parle de **déclaration**
- accéder à sa valeur
- changer sa valeur, on parle d'**assignation**

Les noms de variables sont le plus souvent composés de lettres, de chiffres et du symbole `_`, ils doivent commencer par une lettre.

C

III.2 État d'exécution

L'ensemble des variables et des valeurs auxquelles elles sont associées à un moment de l'exécution d'un programme est un état d'exécution. Les instructions de déclaration et d'assignation modifient ainsi l'état.

Suite aux instructions

```
C int a = 3;
int b = 2;
```

l'état est alors

variable	valeur
a	3
b	2

Si on exécute ensuite l'instruction

```
C b = a;
```

il devient

a	3
b	3

Dans la suite, on pourra noter $a=3$, $b=3$ un tel état. Ici, le $=$ permet une notation intuitive.

III.3 Instructions et blocs

Les programmes qu'on va écrire vont maintenant être des suites d'instructions, comme

```
C int x = 3;
int y = x + 2;
x = y;
```

Ces instructions sont regroupées ensemble sous une notion de blocs. Suivant les langages, ces blocs sont plus ou moins explicites.

Un bloc est délimité par des accolades. Chaque bloc définit sa propre notion de portée.

```
C {
    int x = 3;
    int y = 2;

    x = y + 2;
    { // bloc imbriqué
        int y = 4;
    } // la portée de ce y s'arrête ici

    x = y + 2; // ici y n'est pas celui du bloc imbriqué
}
```

Derrière cette notion de bloc on trouve une notion implicite qui est celle de **flot d'exécution** et qui régit la manière dont les instructions sont exécutées. En accord avec la manière dont on écrit les programmes, les instructions sont exécutées de haut en bas. L'état d'exécution est alors modifié le long du flot.

Une analogie qui sera utile dans la suite est de voir l'état d'un programme comme un train qui circule à travers les rails du flot de contrôle, en rajoutant ou modifiant des wagons à chaque instruction présentée comme un arrêt. Cette image assez naïve permet de faire passer une idée très importante qui est celle de l'ordre d'évaluation et du contexte associé.



Un exemple important est celui de l'échange du contenu de deux variables `a` et `b`. On ne peut pas écrire

```
○ b = a;
  a = b;
```

car comme on vient de le voir, la valeur de `b` est déjà perdue après la première assignation. Une solution consiste alors à introduire une variable temporaire qui ne va servir que de support pour pouvoir garder la valeur de `b`. On écrira ainsi :

```
// ici, on ne peut pas le faire de manière
// générique, cela dépend du type de a et b.
○ int temp = b;
  b = a;
  a = temp;
```

III.4 Portée

Une variable a une durée de vie, souvent, elle n'existe que localement dans un programme. Par exemple, uniquement au sein de la fonction ou du bloc dans laquelle on l'a définie. On parle de portée d'une variable pour désigner l'ensemble des instructions dans lesquelles ont peut y accéder.



III.5 Manipuler des données composées mutables

Les types de données mutables permettent de modifier leur valeur. Ce sont le plus souvent des **collections**, c'est-à-dire des arrangements structurés de valeurs, comme un tableau qui les arrange en ligne de la première à la dernière. Lorsqu'on peut faire un accès direct à une valeur, on peut alors la modifier comme si c'était une variable. Il peut également être possible de faire des modifications sur la structure de la collection elle-même, comme enlever ou rajouter des éléments.

III.5.i Modification des valeurs des éléments



III.5.ii Partages de valeurs composées entre plusieurs variables

Il y a une différence fondamentale entre écrire

Python

```
a = 3
b = a
a = 2
```

et écrire

Python

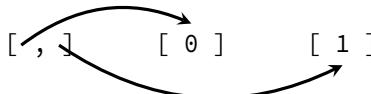
```
a = [ 3 ]
b = a
a[0] = 2
```

Dans le premier cas, lors de l'assignation `b = a`, il n'y a pas de lien entre la valeur 3 dans `a` et celle dans `b`. Dans le second cas, après `b = a`, on a `b` et `a` qui pointent vers le même tableau. On notera `b, a = [3]` cet état. Ainsi, quand on exécute l'instruction `a[0] = 2` on modifie le tableau qui est également associé à `b`. Donc `a` et `b` sont associées au tableau `[2]`.

En fait, quand on écrit `a = [3]`, on ne stocke pas la valeur du tableau dans `a`. On crée le tableau `[3]` en mémoire et on place dans `a` une référence vers ce tableau. Quand on exécute alors `b = a`, on place dans `b` une autre référence mais vers le même tableau. Une manière simple de voir cela est de représenter les références par des flèches, ainsi l'état pourrait en fait se représenter ainsi :



Si on considère un tableau de tableau comme `[[0], [1]]` on aura alors dans les cases du tableau principal des références vers les deux tableaux `[0]` et `[1]` :

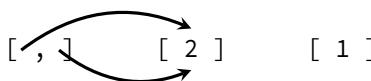


Ainsi, si on écrit

Python

```
t = [ [ 0 ], [ 1 ] ]
t[1] = t[0]
t[1][0] = 2
```

on aura alors les deux références qui pointent vers le même tableau, et quand on modifie `t[1][0]`, on modifie donc forcément `t[0][0]` également pour aboutir à l'état :



On peut remarquer que plus rien ne fait référence ici au tableau `[1]`. Il y a un mécanisme dans Python qui détecte cela et qui supprime de la mémoire le tableau.

Ce qui vient d'être dit pour les tableaux en Python est encore vrai avec le dictionnaires, et plus généralement avec la plupart des données.

III.5.iii Modification de la structure des données composées

Pour parler de modification de tableaux ou de dictionnaires, il est nécessaire de comprendre que Python est un langage objet. On se contentera ici de dire qu'un objet est une donnée munie d'opérations de manipulation sur cette donnée.

Ainsi, si `t` est un tableau, on pourra écrire `t.append(x)` pour ajouter la valeur `x` comme une nouvelle case à la fin du tableau. `t.append(x)` est une instruction, elle ne renvoie pas un nouveau tableau avec cette modification mais elle modifie directement `t`. On dit que `.append` est une **méthode** de la **classe** `list` de `t`. Il est tout à fait possible d'utiliser Python sans écrire de classes, mais dans la mesure où c'est un langage objet dans son cœur, on manipulera forcément des objets.

Par exemple, la suite d'instructions :

```
Python | t = []
        | t.append(2)
        | t.append(3)
```

va faire passer l'état de `t = []` à `t = [2]` puis `t = [2, 3]`.

Il existe beaucoup de méthodes pour les types `list` et `dict`. On les verra selon les besoins dans la suite. Il est toutefois possible, et souhaitable, de se référer à la documentation pour avoir une description détaillée de celles-ci.

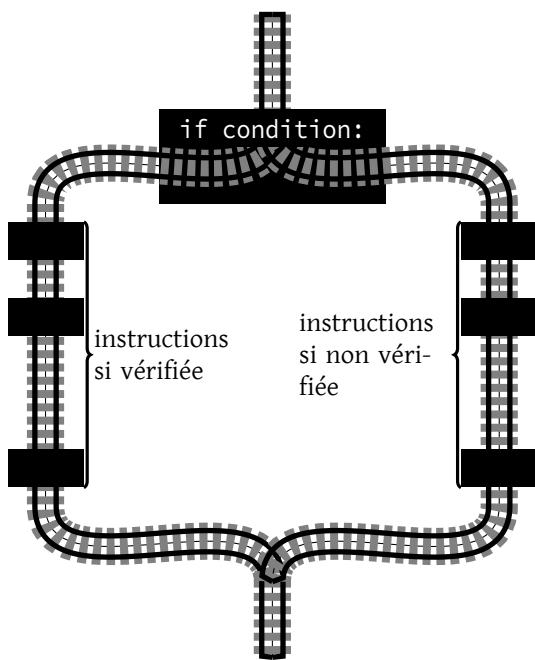
III.6 Instruction conditionnelle

L'instruction conditionnelle va permettre d'enrichir le flot de contrôle avec des branchements. Elle permet d'orienter le flot d'exécution dans un bloc ou dans un autre selon qu'une condition soit vérifiée ou non.

On écrit

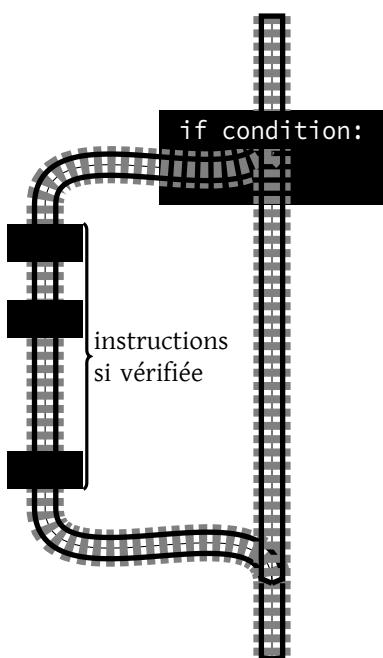
```
○ |
```

Si on reprend l'analogie des trains et des rails, une instruction conditionnelle est un échangeur qui, selon que le train vérifie ou pas la condition va le faire circuler dans une branche ou une autre. Il est important de comprendre qu'une fois une branche exécutée, les flots se rejoignent sur l'instruction qui suit l'instruction conditionnelle.



Il arrive qu'on n'ait rien à faire dans une branche, on peut alors placer un bloc vide, mais, en général, on préfère, quitte à nier la condition, ne pas écrire de `else` comme dans :

Python | `if condition:
 # bloc si la
 # condition est vérifiée
 # le flot saute directement ici si
 # la condition n'est pas vérifiée`



■ **Remarque 3.1** En Python, un bloc vide se note `pass` comme dans le programme suivant :

```
Python if x == 3:
    y = y + 1 # incrémente y si x vaut 3
else:
    pass # ne fait rien sinon
```

qu'on aurait pu écrire :

```
Python if x == 3:
    y = y + 1 # incrémente y si x vaut 3
```

On pourrait être tenter d'écrire l'instruction a priori inoffensive `y = y` dans le bloc du `else`, mais une telle assignation n'est jamais gratuite. En effet, pour certaines données, il peut se produire une duplication coûteuse pour la réaliser et c'est une bonne pratique de ne pas écrire des opérations qui, si pour nous elles ne font rien comme `y = y + 0`, peuvent en fait avoir un coût caché.

Une principe clé de la programmation est la compositionnalité des instructions de gestion de flot : on peut placer des instructions conditionnelles dans le corps d'une des branches. On pourra donc écrire

```
Python if condition1:
    if condition2:
        # bloc si condition 1 et condition 2 sont vérifiées
    else:
        # bloc si condition 1 est vérifiée mais condition 2 ne l'est pas
else:
    if condition2:
        # bloc si condition 1 n'est pas vérifiée et condition 2 l'est
    else:
        # bloc si condition 1 et condition 2 ne le sont pas
```

III.6.i Conditions et opérations sur les booléens

Le rôle des conditions est central dans l'usage des instructions conditionnelles et ainsi c'est très important de bien comprendre le fonctionnement des booléens et de leurs opérations.

Une condition est une formule logique constituée

- de formules atomiques portant sur l'état comme `x == 3`, `x < 2`, ...
- d'opérateurs booléens pour relier ces formules : `not`, `and` ou `or`

III.6.ii Inversion de point de vue par rapport aux mathématiques

En mathématiques, il est courant d'avoir des définitions de valeurs par cas. Par exemple, on pourrait écrire

$$x = \begin{cases} \frac{y}{2} & \text{si } y \text{ pair} \\ \frac{y-1}{2} & \text{sinon.} \end{cases}$$

On pourrait le traduire alors aisément ainsi :

Python

```
if y % 2 == 0:
    x = y / 2
else:
    x = (y-1) / 2
```

Mais en faisant cela, on a inversé le point de vue en plaçant la condition avant l'affectation de x.

III.6.iii Instructions conditionnelles en cascade

III.7 Entrées et sorties

IV Répéter

Jusqu'ici, on a vu des instructions permettant de manipuler l'état d'exécution, éventuellement de manière différente selon sa valeur à l'aide d'instructions conditionnelles. Mais étant donné un état de départ, on connaît déjà les instructions qui seront exécutées et surtout **on connaît leur nombre** qui est majoré par le nombre total d'instructions. En effet, pour reprendre l'analogie des trains, ceux-ci ne font que descendre le long du flot et le nombre d'arrêts rencontrés est majoré par le nombre d'arrêts total.

Cependant, la force principale de l'informatique vient du fait qu'on peut, à l'aide d'un nombre d'instructions fini, avoir la potentialité d'exécuter un nombre arbitrairement grand d'instructions. A cette fin, nous allons introduire un élément fondamental : la notion de répétition ou de boucles.

On pourrait être étonné de la formulation précédente sur ce nombre arbitrairement grand car la mémoire, et même le temps d'exécution en pratique, étant fini, tout est borné. Si on considère un programme qui va calculer la somme des éléments d'un tableau et qu'on sait que la longueur d'un tel tableau est majorée, on pourrait imaginer une énorme boucle conditionnelle

IV.1 Répéter n fois

IV.2 Répéter pour chaque élément

IV.2.i Accumulateur

IV.2.ii Drapeau

IV.3 Répéter tant qu'une condition est vérifiée

IV.4 Choisir la structure de boucle adaptée

IV.5 Boucles imbriquées et dimensionnalité

V Abstraire

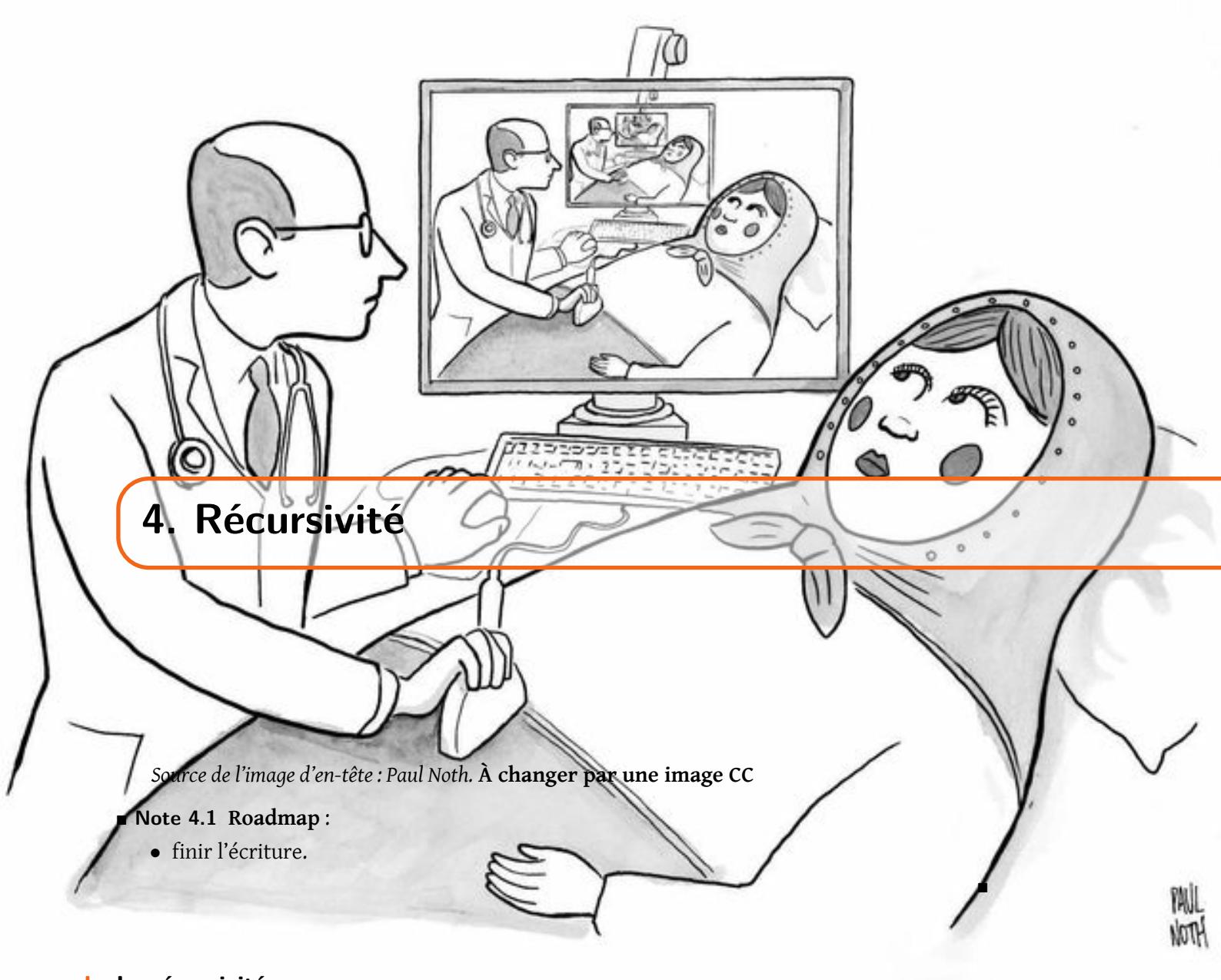
V.1 Définir et appeler des fonctions

V.2 Action d'une fonction et valeur de retour

V.3 Fonctions et portée

V.4 Structurer des programmes avec des fonctions

V.5 Les fonctions comme valeurs



4. Récursivité

Source de l'image d'en-tête : Paul Noth. À changer par une image CC

■ Note 4.1 Roadmap :

- finir l'écriture.

I La récursivité

I.1 Principe et exemples

On dit qu'une fonction est récursive lorsqu'elle va s'appeler elle-même lors de son exécution. Le plus souvent, cet appel sera directement visible depuis le corps de la fonction.

Par exemple, la fonction `fact` suivante permettant de calculer $n!$ est récursive :

```
int fact(int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * fact(n-1);
    }
}
```

On utilise ici un exemple mathématique pour la simplicité des fonctions écrites. En effet, on ne voit rien d'autre dans `fact` que ces appels récursifs et ils correspondent directement à la définition mathématique :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n - 1)! & \text{sinon.} \end{cases}$$

■ **Remarque 4.1** En fait, la définition mathématique se fait plutôt en posant :

$$0! = 1 \quad (n + 1)! = (n + 1) \times n!$$

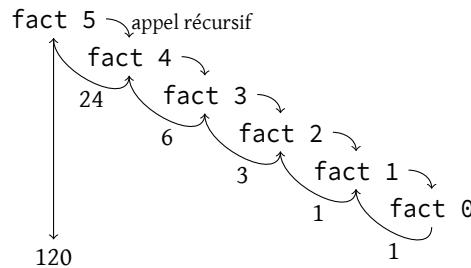
Donc, pour retraduire une telle définition en tant que fonction, il est souvent nécessaire de décaler les rangs.

On remarque tout de suite deux choses :

- il y a un valeur pour laquelle on ne fait aucun appel
- dans les autres cas, quand on fait un appel, l'argument $n - 1$ diminue strictement.

Cela permet d'être sûr qu'on ne va pas s'arrêter indéfiniment. On verra dans la suite comment on peut s'en assurer pour des fonctions plus complexes.

Quand on évalue `fact 5` on va effectuer une série d'appels :



Pour calculer la valeur il faut descendre le long des appels récursifs jusqu'à tomber sur un cas de base puis remonter.

1.2 Implémentation pratique

Comme on vient de le voir, pour calculer une valeur avec une fonction récursive, il est nécessaire de remonter. Pour cela, on a besoin d'interrompre l'évaluation d'une fonction le temps que l'appel récursif se termine, puis de revenir dans l'évaluation là où on en était. On parle alors de contexte d'évaluation ou contexte d'exécution pour l'information qu'il faut sauvegarder afin de reprendre là où on en était. En première approximation, on peut se dire qu'un contexte est la donnée de la position de l'instruction courante, ou de la sous-expression en train d'être évaluée si on raisonne en fonctionnel, ainsi que de l'ensemble des valeurs des variables locales. Il est donc nécessaire de sauvegarder la succession des contextes tout au long des appels récursifs.

On remarque que ce mécanisme est déjà présent pour n'importe quel appel de fonction. La différence principale avec la récursivité, c'est que le nombre d'appels imbriqués à tout moment du programme n'est pas borné à priori, il dépend des paramètres récursifs, et donc il ne peut être connu qu'à l'exécution. On a ainsi besoin d'un mécanisme qui permettent de stocker un nombre quelconque de contextes.

Pour cela, on passe par ce qu'on appelle la pile d'appels. On verra plus tard qu'elle peut être organisée de manière très efficace pour des programmes binaires, mais pour le moment, on se contente d'imaginer que cette pile est vide au début de l'évaluation, puis, à chaque appel, on

empile le contexte courant, on exécute l'appel, une fois la valeur de retour obtenue, on **dépile** le contexte pour reprendre l'exécution là où on en était avant l'appel mais avec cette valeur de retour calculée.

I.3 Programmer en récursif

I.4 Récursivité croisée

II L'arbre d'appels

II.1 Définition

II.2 Complexité en nombre d'appels

II.3 Terminaison

III Récursivité terminale

III.1 Présentation

III.2 Optimisation

III.3 Techniques

IV Types inductifs et induction structurelle

IV.1 Définition naïve des types inductifs



OCaml

5. Type option en OCaml

I Principie

Dans de très nombreux contextes, on a besoin de pouvoir exprimer une notion de partialité en programmant. Cela apparaît en général dans deux cas :

- On veut réaliser une fonction qui ne va pas pouvoir renvoyer une valeur dans tous les cas. Exemple : renvoyer la tête d'une liste chaînée ne fonctionne pas si on passe la liste vide. La fonction est en fait une fonction partielle au sens mathématiques
- On souhaite construire progressivement une donnée et il faut qu'on puisse avoir une notion de valeurs indéterminées. Exemple : on veut remplir au fur et à mesure une grille de Sudoku, à la fin, c'est une matrice de nombres, mais il faut pouvoir gérer les cases vides de manière intermédiaires.

En C, on résout ces questions en utilisant des valeurs comme -1 ou le pointeur nul NULL. En OCaml, le système de types nous pousse à chercher une meilleure solution.

II Syntaxe

La solution en OCaml est très simple, mais demande de la pratique pour l'utiliser à bon escient. Il s'agit, pour tout type 'a de définir un type 'a option permettant de représenter, soit une valeur de type 'a, soit une absence de valeur.

Ce type est défini ainsi :

```
OCaml | type 'a option = None | Some of 'a
```

Cela signifie qu'on a remplacé les valeurs b par des valeurs Some b et que l'absence de valeur est maintenant une vraie valeur None.

Aucune difficulté pour définir une valeur de type option, on se contente d'appeler un des deux constructeurs. Ainsi Some 1 est un int option et Some "test" un string option. La valeur None est polymorphe dans le même sens que [].

Pour manipuler une valeur `o` du type `'a option`, on effectue un filtrage comme pour les autres types somme :

```
OCaml | match o with
| None -> (* ... *)
| Some a -> (* ici on peut accéder au contenu a *)
```

Il est également possible de revenir sur le comportement précédent avec une fonction comme

```
OCaml | let unwrap o =
|       match o with
|       | None -> failwith "None"
|       | Some a -> a
```

Qui permet de déballer une valeur `'a option` en faisant l'hypothèse que ce n'est pas `None`.

■ **Remarque 5.1** Une erreur classique avec le type `option`, qui est la même qu'avec le type `list`, c'est de se focaliser sur `None` en commençant par écrire :

```
OCaml | if o = None
|       then (* .. *)
|       else (* et ici on est bloqué, car o est emballé *)
```

Donc, une règle : si on a une valeur `option`, on effectue un filtrage ! ■

III Utilisation concrète

III.1 Définir une fonction partielle

Il se trouve qu'on a déjà vu un mécanisme permettant de définir une fonction qui peut échouer : les exceptions, notamment avec `failwith`. Si on considère une fonction `f : 'a -> 'b` on a ainsi deux types de valeurs dans le type `'a` :

- Les valeurs `x` qui permettent d'obtenir une valeur `f x` du type `'b`
- Les valeurs `y` pour lesquelles `f y` produit une erreur.

Schématiquement, si note A les valeurs du type `'a`, on a donc $A = A_s \cup A_e$ où l'union est disjointe et A_s sont les valeurs pour lesquelles f ne produit pas d'erreurs et A_e les valeurs produisant des erreurs. Si on prend la définition usuelle de fonctions en mathématiques, A_s est le domaine de f et on considère en fait une **application** de A_s dans B les valeurs du type `'b`.

L'idée avec le type `option`, c'est de transformer toute fonction partielle $f : A \rightarrow B$ en une application $f^* : A \rightarrow B \cup \{\star\}$ où \star est un élément spécial $\star \notin B$ qui correspond à une **valeur erreur**. Cela correspond à une *réification* de l'erreur au rang de valeur. L'application f^* est ainsi définie :

$$f^* : A \rightarrow B \cup \{\star\}$$

$$a \mapsto \begin{cases} f(a) & \text{si } a \in A_s \\ \star & \text{si } a \in A_e \end{cases}$$

La valeur \star est ce qu'on appelle un *puit*, c'est ici qu'on redirige toutes les entrées invalides.

Il n'y a pas de type correspondant directement à l'union $B \cup \{\star\}$ en OCaml et c'est pour cela qu'on utilise le type `'b option` qui nous force à *emballer* un retour.

Ainsi à chaque fois qu'on aurait renvoyé une valeur `b`, on va renvoyer plutôt une valeur `Some b`, et à la place de produire une erreur, on renvoie `None`.

Par exemple, le code suivant :

```
OCaml | let tete l =
        match l with
        | t :: q -> t
        | [] -> failwith "Liste vide"
```

va s'écrire

```
OCaml | let tete_opt l =
        match l with
        | t :: q -> Some t
        | [] -> None
```

III.2 Appeler une fonction partielle

Pour appeler une fonction partielle, il n'y a pas de difficulté, il suffit de faire l'appel de fonction et de manipuler ensuite le type option comme vu plus haut. Cela signifie qu'on effectuera en général l'appel directement dans l'expression d'un filtrage :

```
OCaml | match f a with
      | None -> (* ... *)
      | Some b -> (* ... *)
```

III.3 Données partielles

Pour tout type polymorphe '`a` `t`' on peut en déduire un type partiel '`a` `option t`' dans lequel les valeurs peuvent ne pas être définies. Ainsi, un `int option array` permettra pour chaque case du tableau de :

- soit être définie, et ce sera alors une valeur de la forme `Some k` où `k` est un `int`,
- soit être indéfinie avec la valeur `None`.

Ainsi, une grille de Sudoku qu'on remplirait progressivement aurait le type `int option array array`. Une case valant `None` signifiant qu'elle n'est pas encore remplie. Bien entendu, une fois la grille remplie, toutes les valeurs seraient de la forme `Some k`. On peut ainsi imaginer une transformation de `int option array array` vers `int array array` consistant à passer d'une grille pouvant être partielle, mais pleinement remplie à une grille d'entiers :

```
OCaml | let grille_complete go =
        let g = Array.make_matrix 9 9 0 in
        for i = 0 to 8 do
          for j = 0 to 8 do
            match go.(i).(j) with
            | None -> failwith "Grille incomplète"
            | Some k -> g.(i).(j) <- k
          done
        done;
        g
```

III.4 Chaîne de traitement

On pourrait se dire que le type `option` n'est qu'en fait une version un peu lourde de la production d'erreurs. Mais en fait, la force du type `option` est justement que c'est un type comme

un autre. Cela permet d'effectuer de nombreux traitements en manipulant des fonctions partielles pour définir des données partielles et d'attendre uniquement au dernier moment pour déballer les valeurs. L'idée forte est de manipuler la partialité naturellement.

Un exemple cela peut-être d'avoir une série de traitement à appliquer sur une liste, chaque traitement va possiblement faire apparaître des `None` qu'on va traiter naturellement avec éventuellement une étape finale pour retirer les options :



On remarque que ce style de programmation est de plus en plus privilégié par la bibliothèque standard de OCaml :

- de nombreuses fonctions partielles `f : 'a -> 'b` ont une variante `f_opt : 'a -> 'b option`
- il y a des fonctions manipulant directement des `_opt` comme `List.filter_map` qui effectue un `map` avec une fonction `'a -> 'b option` et filtre les `None` en sortie.
- Le nouveau module `Option` contient beaucoup de fonctions pour se simplifier la vie quand on programme avec les types `option`. **Attention**, cela correspond à une utilisation bien plus pointue de OCaml que dans le reste de ce cours. Notamment, cela fait beaucoup de sens en lien avec l'opérateur `|>` qu'on n'utilise pas ici.



OCaml

6. Exceptions en OCaml

Les exceptions sont un mécanisme présent dans de nombreux langages de programmation afin de permettre à la fois d'indiquer qu'une erreur s'est produite en interrompant l'exécution d'une fonction et aussi de pouvoir, pour l'appelant **rattraper** cette erreur afin de poursuivre l'exécution comme il se doit.

En OCaml, ce mécanisme va permettre, de plus, de pouvoir interrompre le flot de contrôle, notamment de retrouver des mécanismes comme `break`, `continue` et surtout `return`.

Syntaxe des exceptions

Les exceptions forment un type `exn` qui est extensible : il est possible de rajouter un nouveau constructeur pour ce type en écrivant :

```
OCaml (* Pour une exception sans paramètre *)
exception NomDeLException

(* Pour une exception avec paramètre de type t*)
exception NomDeLException of t
```

On peut alors **lancer** cette exception à l'aide de `raise` :

```
OCaml (* Pour une exception sans paramètre *)
raise NomDeLException

(* Pour une exception avec paramètre de valeur v *)
(* Attention aux parenthèses *)
raise (NomDeLException v)
```

Il est possible d'évaluer une expression en permettant d'évaluer une autre expression en cas d'exception :

```
try
  expression
with ExceptionsA attraper -> expression'
```

En fait, ce `with` est un filtrage, un `match`, sur les valeurs du type `exn` mais sans vérification d'exhaustivité. On peut donc effectuer plusieurs cas :

```
try
  expression
with motif1 -> expr1
| motif2 -> expr2
| ...
```

En cas de `try...with` imbriqués, c'est le `try` le plus proche de l'exception qui la rattrape. Si une exception n'est pas rattrapée, elle va produire une erreur qui stoppera l'exécution d'un programme. On peut très bien lancer une exception depuis un `with ... ->`.

II Exceptions pour la gestion d'erreurs

Il s'agit ici de l'utilisation la plus *logique* des exceptions.

On a déjà pu rencontrer des exceptions en OCaml :

```
# let t = [|1;2|];;
val t : int array = [|1; 2|]
# t.(2);;
Exception: Invalid_argument "index out of bounds".
# List.hd [];;
Exception: Failure "hd".
```

Dans les deux cas, ces exceptions sont plutôt le signe d'une erreur de programmation qu'il faut corriger qu'un comportement limite qu'il faudrait prendre en compte. Dans le cas d'une fonction comme `List.hd`, on a également vu qu'il était préférable d'utiliser un type option pour pouvoir gérer l'erreur derrière avec :

```
let hd_opt l =
  match l with
  | [] -> None
  | t :: _ -> Some t
```

Pour autant, il y a des erreurs importantes à gérer en OCaml, ce sont celles qui sont inévitables quand il n'est pas possible de prévoir si un opération va réussir. Citons deux cas :

- la lecture d'un fichier va produire une exception `End_of_file` une fois atteinte la fin de celui-ci;
- des structures de données qui ne fournissent comme seul moyen efficace de savoir si elles sont vides que d'essayer d'en extraire un élément et d'échouer.

Ainsi, dans le premier cas, on pourra écrire une fonction comme :

```
let rec input_all_lines ic =
  try
    let l = input_line ic in
    l :: input_all_lines ic
  with End_of_file -> []
```

Cette fonction prend un `input_channel` et renvoie toutes les lignes qu'il contient sous la forme d'une liste. Le ratrappage d'exception est crucial, c'est lui qui permet d'avoir un cas de base pour la récurrence.

■ **Remarque 6.1 Attention**, on peut être tenté d'écrire :

```
OCaml | let rec input_all_lines ic =
         try
           input_line ic :: input_all_lines ic
         with End_of_file -> []
```

Or, il se trouve que OCaml évalue tous les arguments du constructeur `::` et il le fait de la droite vers la gauche. Le code précédent va donc effectuer des appels récursifs sans jamais lire une seule ligne.

III Programmer avec des exceptions

III.1 Retour prématûré

Si on considère le programme C suivant :

```
C | int recherche_element(int t[],
                        unsigned int nb_elts, int x)
{
    for(int i = 0; i < nb_elts; i++)
        if(t[i] == x)
            return i;
    return -1;
}
```

On peut l'écrire sous la forme suivante en OCaml :

```
OCaml | let recherche_element t x =
       let indice = ref None in
       for i = 0 to Array.length t - 1 do
           if !indice = None && t.(i) = x
               then indice := Some i
       done;
       !indice
```

Cela correspond à ce qu'on aurait fait en C pour se passer du `return` dans la boucle mais cela présente plusieurs problèmes :

- on inhibe les itérations d'une boucle explicitement;
- avec plusieurs boucles imbriquées, c'est peu lisible.

Notons que cette interruption du flux de contrôle est très naturelle dans un langage utilisant beaucoup la récursivité. Le code suivant récursif suivant a la même structure que le code C :

```
OCaml | let recherche_element t x =
       let rec rech_aux i =
           if i = Array.length t
               then None
           else if t.(i) = x then Some i
           else rech_aux (i+1)
```

```
in rech_aux 0
```

En effet, on remarque deux feuilles possibles pour l'arbre d'appels récursifs : soit `None` en fin de tableau, soit `Some i` quand on a trouvé l'élément.

Traduire des programmes impératifs sous cette forme est également peu satisfaisant.

À l'aide des exceptions, on peut retrouver exactement la structure du code C initiale ainsi :

```
exception Trouve of int

let recherche_element t x =
  try
    for i = 0 to Array.length t - 1 do
      if t.(i) = x
      then raise (Trouve i)
    done;
    None
  with Trouve i -> Some i
```

Ou même, en utilisant l'exception `Not_found` prédéfinie :

```
exception Trouve of int

let recherche_element t x =
  try
    for i = 0 to Array.length t - 1 do
      if t.(i) = x
      then raise (Trouve i)
    done;
    raise Not_found
  with Trouve i -> i
```

L'expression `raise (Trouve i)` est ainsi exactement le pendant du `return i;`.

Au lieu d'écrire

```
ocaml
...
return v
...
```

on écrit

```
try
  ...
  raise (Return v)
  ...
failwith "Unreachable"
with Return v -> v
```

III.1.i break et continue

On a déjà vu que `break` permettait de sortir d'une boucle et `continue` de passer à l'itération suivante.

Retrouver ces mécanismes avec OCaml est plus anecdotiques mais on peut le faire. On définit tout d'abord deux exceptions :

```
exception Break
exception Continue
```

Pour un `break` dans une boucle

Au lieu d'écrire

```
for ... do  
...  
break  
...  
done
```

Et pour un `continue`:

Au lieu d'écrire

```
for ... do  
...  
continue  
...  
done
```

on écrit

```
try  
for ... do  
...  
raise Break (* sortie *)  
...  
done  
with Break -> ()
```

on écrit

```
for ... do  
try  
...  
raise Continue  
...  
with Continue -> ()  
done
```


Structures de données

7	Structures de données abstraites et implémentations	55
I	Introduction	
II	Structure de données abstraite	
8	Séquences et ses implémentations : tableaux, listes chaînées	57
I	Structure abstraite séquence ou liste	
II	Implémentations	
III	Implémentations concrètes des Listes chaînées	
IV	Travaux pratiques	
9	Piles et files : structures abstraites et implémentations	85
I	Piles	
II	Files	
10	Arbres	91
I	Arbres binaires	
II	Arbres	
III	Parcours	
IV	Arbres binaires de recherche	
V	Tas	
VI	TP	
11	Graphes	145
I	Graphes orientés	
II	Graphes non orientés	
III	Graphe acyclique connexe	
IV	Parcours	
V	Chemin de plus petit poids	



7. Structures de données abstraites et implémentations

Introduction

On va étudier dans ce chapitre la notion de structures de données. Une structure de données est une manière de stocker des données et d'interagir avec elles. On peut regrouper de grandes classes de structures données en faisant abstraction de la manière dont les données sont stockées en se concentrant uniquement sur les interactions possibles. Cela permet de définir la notion d'une structure de données abstraites.

Dans un second temps, on s'intéresse à la notion d'implémentation d'une structure de données abstraites qui correspond à un choix concret de stockage et donc à une réalisation de l'interface attendue.

Structure de données abstraite

Définition II.1 On appelle *structure de données abstraite* la donnée d'un type t et d'une interface



8. Séquences et ses implémentations : tableaux, listes et chaînes

Structure abstraite séquence ou liste

La structure abstraite *liste*, ou également *séquence* dans des contextes, comme en OCaml, où le terme *liste* fait références aux listes chaînées, est la structure la plus simple pour stocker des données.

Une séquence d'éléments de type t est un type $S(t)$ dont les éléments représentent des valeurs du type t rangées séquentiellement dans un ordre, de la première à la dernière valeur. La vision logique la plus proche de cela est d'imaginer des cases ayant un indice, en commençant en général à l'indice 0, et contenant des valeurs.

II Implémentations

III Implémentations concrètes des Listes chaînées

III.1 En C

III.1.i Représentation et type

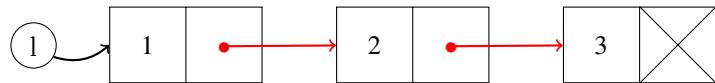
Il existe de nombreuses possibilités d'implémentation des listes chaînées en C. On présente ici les opérations autour d'une implémentation et on discutera ensuite des alternatives.

Dans les limites du programme, on ne présente que des listes permettant de contenir un même type, ici des entiers.

Une liste est ainsi un pointeur sur un maillon et un maillon est un couple (*valeur*, *suivant*) représenté dans une **struct** où *suivant* pointe vers le prochain maillon de la chaîne. Le pointeur nul, de valeur **NULL**, permet ainsi de représenter la liste vide.

```
struct maillon {  
    int valeur;  
    struct maillon *suivant;  
};  
typedef struct maillon maillon;  
typedef maillon *liste;
```

On représentera graphiquement le pointeur nul par une croix et les maillons par des blocs contenant une valeur et un pointeur. Ainsi, le dernier maillon de la liste contient une croix. La liste l correspondant à la valeur qu'on pourrait noter [1, 2, 3] sera représentée ainsi :



III.1.ii Constructeur

On parle de constructeur pour des fonctions qui permettent d'allouer et d'initialiser une valeur d'une structure de donnée. Ici, comme les listes sont des pointeurs sur des maillons, il s'agit uniquement de créer un maillon. Pour cela, on va utiliser `malloc` pour allouer dynamiquement un nouveau maillon.

```
maillon *maillon_creer(int valeur, maillon *suivant)
{
    maillon *m = malloc(sizeof(maillon));
    m->valeur = valeur;
    m->suivant = suivant;
    return m;
}
```

En fait, ce constructeur pourrait être découpé en deux parties : l'allocation qui va se charger de récupérer un emplacement mémoire pour le maillon et l'initialisation qui va attribuer des valeurs.

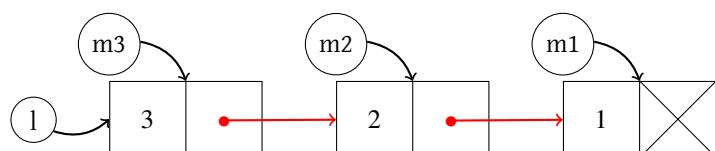
```
maillon *maillon_allouer()
{
    return malloc(sizeof(maillon));
}

void maillon_initialiser(maillon *m, int valeur, maillon *suivant)
{
    m->valeur = valeur;
    m->suivant = suivant;
}

maillon *maillon_creer(int valeur, maillon *suivant)
{
    maillon *m = maillon_allouer();
    maillon_initialiser(m, valeur, suivant);
    return m;
}
```

On peut alors commencer à créer des listes en enchainant les maillons :

```
maillon *m1 = creer_maillon(1, NULL); // pas de suivant, c'est le dernier maillon
maillon *m2 = creer_maillon(2, m1);
maillon *m3 = creer_maillon(3, m2);
liste l = m3; // la liste pointe sur le premier maillon
```



Comme le pointeur sur le premier maillon suffit ici, on aurait pu directement écrire :

```
C | liste l = creer_maillon(3,
                           creer_maillon(2,
                                         creer_maillon(1, NULL)));
```

III.1.iii Déstructeur

Pour détruire un maillon, il suffit de libérer l'espace qu'on lui a attribué.

```
C | void maillon_detruire(maillon *m)
  {
    free(m);
  }
```

Pour détruire une liste, on va par contre avoir besoin de parcourir l'ensemble des maillons qui la constitue. Comme pour les autres parcours, on a alors deux choix :

- **parcours récursif** on a un cas de base quand la liste est vide et dans le cas général, un éventuel appel récursif sur le pointeur suivant.

```
C | void liste_detruire(liste l)
  {
    if (l != NULL)
    {
      liste_detruire(l->suivant);
      maillon_detruire(l); // Attention à l'ordre pour l->suivant
    }
  }
```

- **parcours impératif** on boucle tant que la liste est non nulle. On fait ici attention à ne pas accéder à `->suivant` après avoir libéré le maillon.

```
C | void liste_detruire(liste l)
  {
    while (l != NULL)
    {
      liste suivante = l->suivant;
      maillon_detruire(l);
      l = suivante;
    }
  }
```

III.1.iv Ajout et suppression en tête

Pour ajouter ou supprimer un maillon en tête de la liste, on va avoir besoin de modifier le pointeur vers le premier maillon. Pour cela, on a deux approches possibles :

- passer un pointeur vers la liste elle-même, c'est-à-dire un pointeur sur un pointeur sur un maillon, ce qui en C aura le type `maillon **`, qui s'écrit aussi `liste *`. On aura alors le prototype :

```
C | void liste_ajout_en_tete(liste *l, int valeur);
```

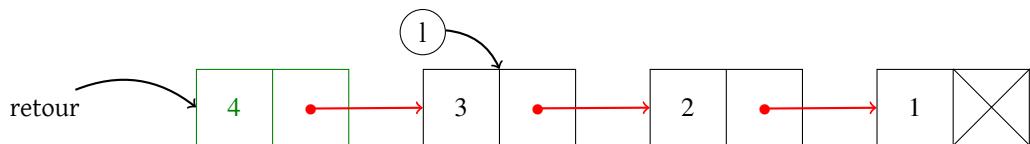
- renvoyer un pointeur vers le nouveau premier maillon. On aura alors le prototype :

```
C | liste liste_ajout_en_tete(liste l, int valeur);
```

On présente ici les versions renvoyant une nouvelle liste :

```
C | liste liste_ajout_en_tete(liste l, int valeur)
{
    maillon *m = maillon_creer(valeur, l);
    return m;
}
```

Si `l` est la liste précédente contenant 3, 2, 1 et qu'on ajoute 4 en tête, on va donc directement créer un nouveau maillon et renvoyer un pointeur vers celui-ci.



```
C | liste liste_suppr_en_tete(liste l)
{
    assert(l != NULL);
    liste queue = l->suivant;
    maillon_detruire(l); // Attention, on détruit juste le maillon, pas la liste
    return queue;
}
```

Si `l` est la liste précédente contenant 3, 2, 1 et qu'on supprime le maillon de tête, on va renvoyer un pointeur sur le second maillon. **Attention**, ici le pointeur initial `l` est devenu invalide.



III.1.v Longueur de la liste

On présente ici le calcul de la longueur comme exemple de parcours de la liste. C'est encore très proche du parcours effectué dans le destructeur.

```
C | int liste_longueur(liste l)
{
    int longueur = 0;
    while(l != NULL)
    {
        longueur = longueur + 1;
        l = l->suivant;
    }
    return longueur;
}
```

III.1.vi Accès au *nième* maillon de la liste

On effectue un parcours similaire pour accéder au *nième* maillon.

```
C maillon *liste_nieme(liste l, int n)
{
    while(n > 0)
    {
        assert(l != NULL);
        l = l->suivant;
        n = n-1;
    }
    assert(l != NULL);
    return l;
}
```

Toujours avec un programme similaire, on peut chercher un maillon avec sa valeur :

```
C maillon *liste_recherche(liste l, int x)
{
    while(l != NULL && l->valeur != x)
    {
        l = l->suivant;
    }
    return l;
}
```

Ici, pas besoin d'asserts, en cas d'échec de la recherche, on renvoie un pointeur nul.

III.1.vii Ajout/Suppression ailleurs qu'en tête

Pour ajouter ou supprimer ailleurs qu'en tête, il est nécessaire de pouvoir repérer précisément un maillon. Pour cela, on peut le faire :

- par son indice, celui utilisé dans `liste_nieme`;
- par sa valeur, avec une recherche ;
- ou encore directement par un pointeur sur le maillon.

Une fois le maillon ajouté/supprimé, on peut procéder comme pour un ajout/suppression en tête, cependant il va falloir reconnecter le pointeur suivant du maillon précédent. Pour cela deux choix :

- soit on considère qu'on ajoute après, auquel cas on dispose du précédent
- soit on considère qu'on ajoute avant/supprime et pour cela on effectue une boucle pour déterminer le maillon précédent.

```
C void liste_ajout_apres(liste l, maillon *m, int x)
{
    m->suivant = liste_ajout_en_tete(m->suivant, x);
}

void liste_ajout_avant(liste l, maillon *m, int x)
{
    liste prec = NULL;

    while(l != m)
    {
        assert(l != NULL);
        prec = l;
        l = l->suivant;
    }
}
```

```

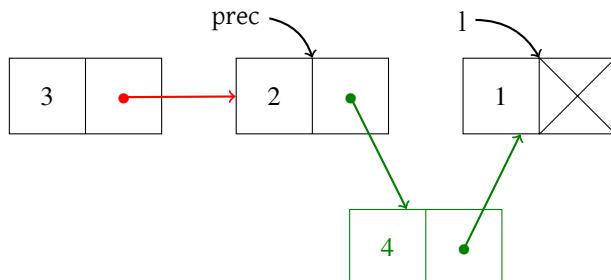
    prec->suivant = liste_ajout_en_tete(m, x);
}

```

Si on reprend la liste `l` contenant `3, 2, 1` et qu'on souhaite insérer la valeur `4` avant la valeur `1`, on va passer un pointeur `m` vers le maillon contenant `1` et effectuer un parcours jusqu'à avoir `prec` et `l` dans la configuration :



On dispose alors des pointeurs permettant de réaliser l'ajout :



```

void liste_suppr_non_en_tete(liste l, maillon *m)
{
    liste prec = NULL;

    while(l != m)
    {
        assert(l != NULL);
        prec = l;
        l = l->suivant;
    }
    prec->suivant = liste_suppr_en_tete(m);
}

```

En fait, lorsqu'on regarde le parcours précédent, on remarque deux points :

- comme on a donné le maillon concerné, le parcours a uniquement pour but de repérer le maillon qui le précède;
- on pourrait se contenter d'utiliser uniquement un pointeur sur le maillon précédent lors du parcours car le maillon qui le suit est accessible avec `->suivant`.

III.1.viii Autres implémentations

Une autre implémentation standard consiste à *cacher* le pointeur sur le premier maillon, ce qui permet de donner également un pointeur sur le dernier maillon. En effet, le pointeur sur le dernier maillon permet de réaliser un ajout en fin de liste en $O(1)$ car il n'y a pas besoin de parcourir la liste pour faire cet ajout.

On obtient alors un type comme :

```

struct maillon {
    int valeur;
    struct maillon *suivant;
};

```

```
typedef struct maillon maillon;

struct liste {
    maillon *premier;
    maillon *dernier;
};

typedef struct liste liste;
```

Cette implémentation est l'occasion de se poser la question sur la répartition entre pile et tas pour les données. On peut légitimement penser que les listes ici ne sont que des couples de pointeurs, et qu'ainsi les passer par copie est léger comparativement à la complexité induite par une allocation sur le tas.

III.2 En OCaml

III.2.i Cas des 'a list

Tout d'abord, il faut se rendre compte que le type par défaut

```
type 'a list = (::) of 'a * 'list | []
```

fait intervenir des maillons et des pointeurs. La différence principale avec C est qu'on ne peut pas changer la valeur des pointeurs.

III.2.ii Type des listes chaînées

Pour retrouver la richesse du type précédent, on peut définir un type comme :

OCaml

```
type 'a maillon = {
    mutable valeur : 'a;
    mutable suivant : 'a liste
} and 'a liste = Vide | Lien of 'a maillon
```

Ici, le type est une traduction directe du type précédent. On remarque que les types sont mutuellement récursifs car un maillon contient une liste. Le type somme '`'a liste`' ressemble fortement à un pointeur qui peut être nul ou pointer sur un maillon. On remarque qu'on aurait pu aussi se contenter d'écrire le type suivant :

OCaml

```
type 'a maillon = {
    mutable valeur : 'a;
    mutable suivant : 'a maillon option
}

type 'a liste = 'a maillon option
```

Mais on va préférer le premier type qui a l'avantage de permettre de bien faire apparaître la structure.

III.2.iii Ajout et suppression en tête

Pour rajouter un maillon en tête, on peut écrire :

OCaml

```
let cons x l = Lien { valeur = x; suivant = l}
```

On remarque que la fonction est beaucoup plus simple que celle en C car l'allocation est automatique et l'initialisation se fait naturellement dans la syntaxe. Cette fonction renvoie une

nouvelle liste, on aurait pu aussi rajouter une référence pour les listes afin de permettre de les rendre modifiables, c'est la même discussion que dans la partie précédente.

La suppression en tête, cela revient à renvoyer la queue de la liste comme le fait `List.tl`:

```
OCaml | let tl l = match l with
      | Vide -> failwith "Liste vide"
      | Lien { valeur = _; suivant = q} -> q
```

On remarque le filtrage imbriqué `| Lien { valeur = t; suivant = q } ->` qui correspond au filtrage `| t :: q` du type `a list`.

III.2.iv Exemple de parcours sans modification : calcul de la longueur

Vu les remarques précédentes, il n'est pas étonnant que le parcours d'une liste de manière récursive soit très proche de ce qu'on a déjà pu voir avec les listes de base.

```
OCaml | let rec longueur l = match l with
      | Vide -> 0
      | Lien { valeur = _; suivant = q } -> 1 + longueur q
```

On en déduit de même une fonction renvoyant un maillon par son indice :

```
OCaml | let rec nieme l n = match l with
      | Vide -> failwith "Liste vide"
      | Lien m -> if n = 0 then m else nieme m.suivant (n-1)
```

■ Remarque 8.1 Il est possible de réécrire la fonction précédente en permettant à la fois de donner un nom, ici `m`, au maillon et de faire un filtrage sur ce qu'il contient. Pour cela, on utilise le mot clé `as` en OCaml :

```
OCaml | let rec nieme l n = match l with
      | Vide -> failwith "Liste vide"
      | Lien ({ valeur=_; suivant = q} as m) ->
          if n = 0 then m else nieme q (n-1)
```

III.2.v Exemple de parcours avec modification : ajout d'une maillon en fin de liste

On va montrer un exemple de modification de liste en rajoutant un maillon en fin d'une liste non vide. Ici, on effectue un parcours jusqu'à tomber sur le dernier maillon auquel on rajoute le nouveau à la suite.

```
OCaml | let rec ajout_fin l x =
      match l with
      | Vide -> failwith "Liste vide"
      | Lien m ->
          if m.suivant = Vide
          then m.suivant <- Lien { valeur = x; suivant = Vide }
          else ajout_fin m.suivant x
```

III.2.vi Raffinement pour permettre l'ajout à la fin en temps constant

Le programme précédent est à comparer au programme suivant :

```
OCaml | let rec ajout_fin l x =
        match l with
        | [] -> [x]
        | t::q -> t :: ajout_fin q x
```

On n'a pas l'impression d'avoir vraiment gagné en expressivité ou en efficacité.

Cependant, on peut se dire qu'en changeant le type '`'a liste`' on peut tirer partie des maillons pour obtenir un ajout en fin de liste en temps constant. Le programme suivant présente une interface permettant de le faire.

```
OCaml | type 'a maillon = {
          mutable valeur : 'a;
          mutable suivant : 'a maillon_ptr
        }
and 'a maillon_ptr = 'a maillon option
and 'a liste = {
          mutable premier : 'a maillon_ptr;
          mutable dernier : 'a maillon_ptr
        }

let liste_vide () = { premier = None; dernier = None }

let ajout_debut l x =
  l.premier <- Some { valeur = x; suivant = l.premier };
  if l.dernier = None
  then l.dernier <- l.premier

let suppr_debut l =
  match l.premier with
  | None -> failwith "Liste vide"
  | Some { valeur = _; suivant = l' } ->
    l.premier <- l'

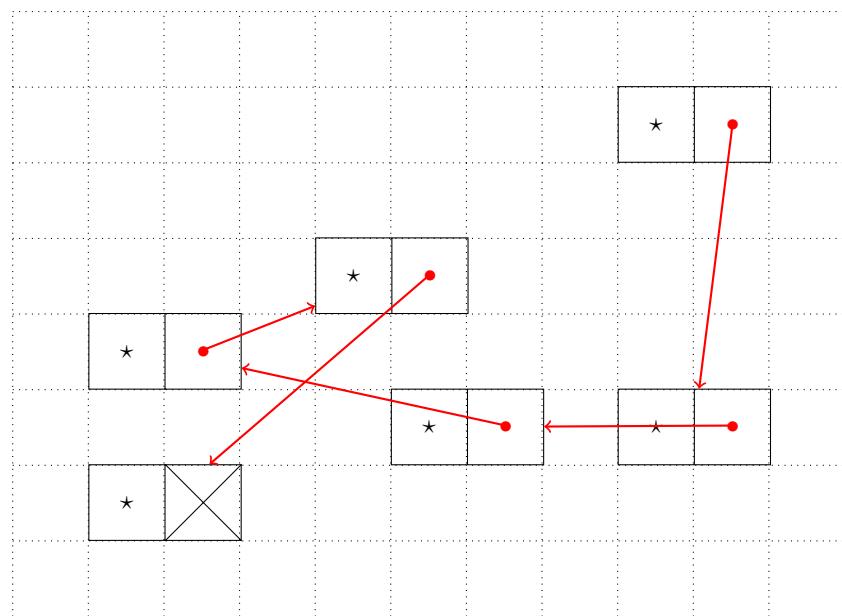
let ajout_fin l x =
  let m = { valeur = x; suivant = None } in
  (match l.dernier with
  | Some m' -> m'.suivant <- Some m
  | None -> ());
  l.dernier <- Some m;
  if l.premier = None
  then l.premier <- l.dernier
```

Quelques remarques sur ce programme :

- comme on change directement `premier` et `dernier`, il est nécessaire de générer une nouvelle liste vide, ce qui est assuré ici par le paramètre `()` ;
- les pointeurs sont représentés par des options dans des champs mutables ;
- afin de préserver l'intégrité des deux pointeurs, on est obligé de gérer les cas où il n'y a qu'un seul maillon ;
- attention à la priorité du cas de filtrage sur `;` qui oblige à mettre des parenthèses dans `ajout_fin`.

III.3 Structure de la mémoire

En mémoire, les maillons d'une liste chaînée, comme celle vue en C, sont sur le tas et de manière désorganisée. Cela signifie qu'il n'y a aucune raison que deux maillons proches dans une liste soient proches en mémoire.



Or, les processeurs optimisent la gestion de la mémoire à l'aide d'un cache qui, au lieu de n'accéder qu'à une seule valeur située à une adresse, va charger une zone mémoire autour de cette adresse. Ceci encourage une cohérence spatiale dans l'organisation de la mémoire afin de profiter au maximum de cette mise en cache.

Une stratégie pour réaliser des listes chaînées efficacement peut consister à allouer un tableau de maillons et à gérer ensuite les allocations parmi cette réserve.

IV Travaux pratiques

On présente ici deux énoncés de travaux pratiques en C en lien avec ce chapitre.

IV.1 Tableaux non statiques et tableaux dynamiques

On va définir ici une structure de donnée pour gérer des tableaux dont la taille ne sera connu qu'à l'exécution. Pour simplifier, ce seront des tableaux d'entiers `int` mais la méthode s'adapte naturellement pour des tableaux de n'importe quoi.

IV.1.i Type array

On définit le type suivant :

```
struct array {
    int *elements;
    unsigned int size;
};
typedef struct array array;
```

Un `array` est donc une structure qui contient :

- un pointeur vers un tableau `elements` d'entiers
- un entier indiquant la taille de ce tableau

Remarque importante quand on copie un pointeur, on copie uniquement un entier qui est l'adresse pointée. Ainsi, quand on copie un array, ce n'est finalement qu'un couple d'entiers qu'on copie. C'est pourquoi, dans la suite, on passe tous les array par valeur. C'est-à-dire que pour définir une fonction de recherche d'un élément dans un array on va écrire :

```
C int search(array t, int x)
{
    for(int i = 0; i < t.size; i++)
    {
        if(t.elements[i] == x)
        {
            return i;
        }
    }

    return -1;
}
```

Quand on va appeler la fonction, on va avoir une copie de l'argument, c'est-à-dire du array, mais le tableau elements lui sera toujours à la même place. L'alternative serait de passer les valeurs array par pointeurs comme dans la variante suivante :

```
C int search(array *t, int x)
{
    for(int i = 0; i < t->size; i++)
    {
        if(t->elements[i] == x)
        {
            return i;
        }
    }

    return -1;
}
```

On n'a rien à gagner au surplus de complexité induit par cela. L'unique avantage serait de permettre de modifier les paramètres de la structure passée en argument, mais on réservera ça aux fonctions qui le nécessitent.

IV.1.ii Allocation, Initialisation, Libération

Pour les fonctions suivantes, on fera usage de malloc et free.

Question IV.1 Écrire une fonction de prototype array array_alloc(unsigned int size) qui alloue un tableau de size entiers avec malloc et renvoie le array pointant dessus.

Conformément à ce qui est écrit au-dessus, on peut créer un array en variable locale et le renvoyer avec return, ce qui compte c'est la mémoire pointée.

Démonstration.

```
C array array_alloc(unsigned int size)
{
    array a;
```

```
a.elements = (int *)malloc(sizeof(int) * size);
a.size = size;
return a;
}
```

Question IV.2 Écrire une fonction de prototype void array_free(array t) qui libère le tableau pointé par t avec free. A partir de ce moment, on ne peut plus utiliser cette adresse sans faire d'erreurs.

Démonstration.

```
void array_free(array a)
{
    free(a.elements);
}
```

Question IV.3 Le tableau alloué n'est pas initialisé, écrire une fonction de prototype array array_make(unsigned int size, int def) qui alloue un tableau et initialise toutes les valeurs de celui-ci à def.

Démonstration.

```
array array_make(unsigned int size, int def)
{
    array a = array_alloc(size);
    for(int i = 0; i < size; i++)
    {
        a.elements[i] = def;
    }
    return a;
}
```

Question IV.4 Il est possible de mesurer le temps pris par un programme à l'aide de la fonction time :

```
$> time ./main
./main 0.38s user 0.00s system 99% cpu 0.379 total
```

Comparer le temps d'exécution d'un programme qui

- alloue puis libère 100 fois un tableau d'un million d'entiers
- alloue *en initialisant à 0* avec make puis libère 100 fois un tableau d'un million d'entiers.

Que peut-on en conclure?

Démonstration.

On se rend compte que l'allocation est presque instantanée alors que l'initialisation prend un temps linéaire en la taille des données. Ceci est cohérent avec le fait que la mémoire est allouée de manière paresseuse. On s'en rend d'autant plus compte en examinant l'empreinte mémoire des programmes.



- **Remarque 8.2** Il existe d'autres fonctions que `malloc` comme

```
void *calloc(size_t nmemb, size_t size);  
void *realloc(void *ptr, size_t size);
```

qui permettent :

- pour `calloc` d'allouer un tableau de `nmemb` membres qui sont chacun de taille `size` et **d'initialiser les octets à 0**. Cette fonction prend donc un temps linéaire en le nombre d'octets.
- pour `realloc` de déplacer en mémoire un tableau en changeant sa taille (en plus ou en moins). Si jamais il y a de la place on ne bouge pas le tableau. C'est donc forcément plus efficace que de le faire à la main.

Ces deux fonctions sont **hors programme!**

**IV.1.iii Affichage, copie**

Question IV.5 Écrire une fonction `void array_print(array a)` qui affiche `[a1,a2,...,an]` si `a` contient les éléments de `a1` à `an`.

Remarque cette fonction vous sera utile pour tester ce que vous avez fait dans la suite.

*Démonstration.*

```
void array_print(array a)  
{  
    printf("[");  
    for(int i = 0; i < a.size; i++)  
    {  
        printf("%d", a.elements[i]);  
        if(i < a.size - 1)  
            printf(",");  
    }  
    printf("]\n");  
}
```



Question IV.6 Écrire une fonction de prototype

```
void array.blit(array src, int src_start,  
                array dst, int dst_start, int len)
```

qui copie `len` éléments depuis le tableau `src` à partir de l'indice `src_start` dans le tableau `dst` à partir de l'indice `dst_start`.



Remarque on a déjà écrit cette fonction en OCaml, vous pouvez vous en inspirer.

Démonstration.

```
void array.blit(array src, int src_start,
                array dst, int dst_start, int len)
{
    for(int i = 0; i < len; i++)
        dst.elements[i + dst_start] =
            src.elements[i + src_start];
```

IV.1.iv Implémentation d'une pile non bornée

Question IV.7 Écrire une fonction de prototype void `array_push(array *a, int x)` qui rajoute `x` à la fin du array `a`, c'est-à-dire :

- on va allouer un nouveau array `b` dont le tableau contient `a->size+1` éléments
- on va copier les anciennes valeurs de `*a` dans `b`
- écrire `x`
- libérer `*a`
- remplacer avec `*a = b`.

Remarque On peut tout à faire écrire `*a = b` pour que `a` ait les mêmes valeurs que `b` en dehors de la fonction ;

Rappel pour une struct par pointeurs, on utilise `->` pour accéder aux champs. Donc `a->elements` et `a->size` ici.

Remarque Faites des tests comme en créant un tableau de taille 0 et en empilant les entiers de 1 à 10 et en affichant le résultat. Faites les tests avec `-fsanitize=address`.

Démonstration.

```
void array.push(array *a, int x)
{
    array b = array_alloc(a->size+1);
    array.blit(*a, 0, b, 0, a->size);
    b.elements[a->size] = x;
    array_free(*a);
    *a = b;
}

// Note qu'on aurait pu proposer une interface persistante pour push/pop ainsi
// pas de pointeurs mais on renvoie le nouveau array
array array.push_2(array a, int x)
{
    array b = array_alloc(a.size+1);
    array.blit(a, 0, b, 0, a.size);
    b.elements[a.size] = x;
    // pas de raison de faire de free ici de a, c'est
    // à l'appelant de gérer.
    return b;
}
```

Question IV.8 Écrire une fonction `int array_pop(array *a)` qui retire le dernier élément du tableau `a` et renvoie sa valeur. Il faudra donc :

- allouer un nouveau `array b` dont le tableau contient `a->size-1` éléments
- copier les anciennes valeurs sauf une
- récupérer dans une variable `x` la valeur de la dernière case de `a` (*parce qu'il n'est pas possible d'y accéder après l'étape suivante*)
- libérer l'ancien tableau
- remplacer avec `*a = b`.

Démonstration.

```
C int array_pop(array *a)
{
    array b = array_alloc(a->size-1);
    array_blit(*a, 0, b, 0, a->size-1);
    int x = a->elements[a->size-1];
    array_free(*a);
    *a = b;
    return x;
}
```

IV.1.v Tableaux dynamiques

Reprendre toutes les questions précédentes en changeant la structure de donnée pour permettre de faire des tableaux dynamiques. Il faudra ainsi avoir deux entiers : `int p_size` qui contiendra la taille *physique* en mémoire et `int l_size` qui contiendra la taille *logique* c'est-à-dire les éléments signifiants stockés.

Quand on a besoin de realloquer la mémoire, on choisira de doubler la taille physique. Le pop ne libérera jamais de taille physique.

Démonstration.

```
C #include <stdio.h>
#include <stdlib.h>

struct array {
    int *elements;
    unsigned int p_size;
    unsigned int l_size;
};

typedef struct array array;

array array_alloc(unsigned int size)
{
    array a;
    a.elements = (int *)malloc(sizeof(int) * size);
    a.p_size = size;
    a.l_size = 0;
}
```

```
        return a;
    }

void array_free(array a)
{
    free(a.elements);
}

array array_make(unsigned int size, int def)
{
    array a = array_alloc(size);
    for(int i = 0; i < size; i++)
    {
        a.elements[i] = def;
    }
    a.l_size = size;
    return a;
}

void array_print(array a)
{
    printf("[");
    for(int i = 0; i < a.l_size; i++)
    {
        printf("%d", a.elements[i]);
        if(i < a.l_size - 1)
            printf(",");
    }
    printf("]\n");
}

void array.blit(array src, int src_start,
                array dst, int dst_start, int len)
{
    for(int i = 0; i < len; i++)
        dst.elements[i + dst_start] =
            src.elements[i + src_start];
}

void array.push(array *a, int x)
{
    if (a->l_size < a->p_size)
    {
        a->elements[a->l_size] = x;
        a->l_size += 1;
    }
    else
    {
        array b = array_alloc(2*a->l_size);
        array.blit(*a, 0, b, 0, a->l_size);
        b.elements[a->l_size] = x;
        b.l_size = a->l_size + 1;
        array_free(*a);
        *a = b;
    }
}
```

```

}

int array_pop(array *a)
{
    a->l_size -= 1;
    return a->elements[a->l_size];
}

int main(void)
{
    array a = array_alloc(0);
    for(int i = 0; i < 10; i++)
        array_push(&a, i);
    for(int i = 0; i < 3; i++)
        printf("Pop %d\n", array_pop(&a));
    array_print(a);
    array_free(a);
}

```



IV.2 Listes chaînées

IV.2.i Listes simplement chaînées

On va définir des fonctions sur les listes chaînées en utilisant le type suivant :

```

struct maillon {
    int valeur;
    struct maillon *suivant;
};
typedef struct maillon maillon;

struct liste {
    maillon *premier;
    maillon *dernier;
};
typedef struct liste liste;

```

■ Remarque 8.3 Invariants à maintenir

- Dans un maillon, `suivant` est soit `NULL` si c'est le dernier maillon de la chaîne, soit un pointeur vers le maillon qui le suit.
- Si la liste est vide alors `premier` et `dernier` valent `NULL`. Sinon, ils pointent respectivement sur le premier et le dernier maillon de la chaîne.



Question IV.9 Écrire une fonction de prototype

C | `liste liste_vide();`

qui renvoie la liste vide. Ici, on pourrait dire **une** liste vide mais cela correspond à une unique valeur de la structure `liste`.



Démonstration.

```

    liste liste_vide()
{
    liste l;
    l.premier = NULL;
    l.dernier = NULL;
    return l;
}

```



Question IV.10 Écrire une fonction de prototype

où

```
void liste_affiche(liste l);
```

qui affiche le contenu de la liste sous la forme [1,2,3].



Démonstration.

```

void liste_affiche(liste l)
{
    putchar('[');
    maillon *m = l.premier;
    while(m != NULL)
    {
        printf("%d", m->valeur);
        if(m->suivant != NULL)
            putchar(',');
        m = m->suivant;
    }
    putchar(']');
    putchar('\n');
}

```



Question IV.11 Écrire une fonction de prototype

où

```
maillon *maillon_creer(int valeur, maillon *suivant);
```

qui alloue et initialise un nouveau maillon.



Démonstration.

```

maillon *maillon_creer(int valeur, maillon *suivant)
{
    maillon *m = malloc(sizeof(maillon));
    m->valeur = valeur;
    m->suivant = suivant;
    return m;
}

```

Question IV.12 Écrire une fonction de prototype

C `void maillon_detruire(maillon *m);`

qui détruit, c'est-à-dire libère, la mémoire associée à un maillon.

Écrire une fonction de prototype

C `void chaine_detruire(maillon *m);`

qui détruit la chaîne de maillon accessible depuis le maillon m.

En déduire une fonction de prototype

C `void liste_detruire(liste l);`

qui détruit la chaîne pointée par une liste.

Démonstration.

```
void maillon_detruire(maillon *m)
{
    free(m);
}

void chaine_detruire(maillon *m)
{
    while(m != NULL)
    {
        maillon *suivant = m->suivant;
        maillon_detruire(m);
        m = suivant;
    }
}

void liste_detruire(liste l)
{
    chaine_detruire(l.premier);
}
```

Question IV.13 Écrire une fonction de prototype

C `int liste_longueur(liste l);`

qui renvoie la longueur de la liste l.

Démonstration.

```
int liste_longueur(liste l)
{
    int longueur = 0;
```

```

maillon *m = l.premier;
while(m != NULL)
{
    longueur += 1;
    m = m->suivant;
}
return longueur;
}

```



Question IV.14 Écrire des fonctions de prototype

○ `int liste_tete(liste l);`
`liste liste_queue(liste l);`

qui renvoient respectivement la tête et la queue d'une liste. **Attention**, ici, contrairement au type vu plus haut, le maillon suivant n'est pas une liste.



Démonstration.

```

int liste_tete(liste l)
{
    assert(l.premier != NULL);
    return l.premier->valeur;
}

liste liste_queue(liste l)
{
    assert(l.premier != NULL);
    liste q;
    q.premier = l.premier->suivant;
    q.dernier = l.dernier;
    return q;
}

```



Question IV.15 Écrire une fonction de prototype

○ `void liste_ajout_en_tete(liste *l, int valeur);`

qui ajoute un maillon en tête de la liste pointée par `l` en $O(1)$. On fera attention au cas où `l` pointe la liste vide.

■ **Note 8.1** Dans cette fonction et les suivantes, on a fait le choix de passer des pointeurs sur des listes pour permettre de modifier les valeurs des pointeurs `premier` et `dernier`. Alternativement, on aurait pu faire en sorte que ces fonctions renvoient des `liste` par copie.



Démonstration.

```
C void liste_ajout_en_tete(liste *l, int valeur)
{
    l->premier = maillon_creer(valeur, l->premier);
    if(l->dernier == NULL) // la liste était vide
        l->dernier = l->premier;
}
```

Question IV.16 Écrire une fonction de prototype

```
C void liste_suppr_en_tete(liste *l);
```

qui supprime un maillon en tête de la liste l supposée non vide en $O(1)$.

On fera attention au cas où on supprime l'unique maillon de la liste.

Démonstration.

```
C void liste_suppr_en_tete(liste *l)
{
    assert(l->premier != NULL);
    maillon *m = l->premier;
    l->premier = m->suivant;
    maillon_detruire(m); // on libère la mémoire
    if(l->premier == NULL) // on a vidé la liste
        l->dernier = NULL;
}
```

Question IV.17 Écrire une fonction de prototype

```
C void liste_ajout_en_fin(liste *l, int valeur);
```

qui ajoute un maillon en fin de la liste l en $O(1)$.

Démonstration.

```
C void liste_ajout_en_fin(liste *l, int valeur)
{
    maillon *m = l->dernier;
    l->dernier = maillon_creer(valeur, NULL);
    m->suivant = l->dernier;
    if(l->premier == NULL) // la liste était vide
        l->premier = l->dernier;
}
```

Question IV.18 Écrire une fonction de prototype

maillon *liste_cherche_maillon(liste l, int valeur);

qui renvoie un pointeur sur le premier maillon de valeur valeur ou renvoie NULL s'il n'y en a pas.

Démonstration.

```
maillon *liste_cherche_maillon(liste l, int valeur)
{
    maillon *m = l.premier;
    while(m != NULL && m->valeur != valeur)
    {
        m = m->suivant;
    }
    return m;
}
```

Question IV.19 Écrire une fonction de prototype

void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur);

qui insère en $O(1)$ un maillon après le maillon m. On fera en sorte que l'insertion soit valide même si m est le dernier maillon de la liste.

Démonstration.

```
void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur)
{
    m->suivant = maillon_creer(valeur, m->suivant);
}
```

Question IV.20 Écrire une fonction de prototype

void

qui ajoute un maillon avant m. Dans le cas où m est le premier ou le dernier maillon, l'insertion sera en $O(1)$.

Démonstration.

```
void liste_ajout_maillon_avant(liste *l, maillon *m, int valeur)
{
    if (m == l->premier)
        liste_ajout_en_tete(l, valeur);
    else if (m == l->dernier)
        liste_ajout_en_fin(l, valeur);
    else
```

```

maillon *prec = l->premier;
while(prec->suivant != m)
{
    prec = prec->suivant;
}
prec->suivant = maillon_creer(valeur, m);
}
}

```

Question IV.21 Écrire une fonction de prototype

↳ `void liste_suppr_maillon(liste *l, maillon *m);`

qui supprime un maillon. Dans le cas où m est le premier maillon, la suppression sera en $O(1)$.

Démonstration.

```

void liste_suppr_maillon(liste *l, maillon *m)
{
    if (m == l->premier)
        liste_suppr_en_tete(l);
    else
    {
        maillon *prec = l->premier;
        while(prec->suivant != m)
        {
            prec = prec->suivant;
        }
        prec->suivant = m->suivant;
        maillon_detruire(m);
    }
}

```

IV.2.ii Listes doublement chaînées

Pour les listes doublement chaînées, on va adapter le type précédent en rajoutant juste un pointeur `precedent` dans les maillons :

```

struct maillon {
    int valeur;
    struct maillon *suivant;
    struct maillon *precedent;
};
typedef struct maillon maillon;

struct liste {
    maillon *premier;
    maillon *dernier;
};
typedef struct liste liste;

```

■ **Remarque 8.4 Invariants à maintenir**

- Dans un maillon, suivant est soit NULL si c'est le dernier maillon de la chaîne, soit un pointeur vers le maillon qui le suit.
- Dans un maillon, précédent est soit NULL si c'est le premier maillon de la chaîne, soit un pointeur vers le maillon qui le précède.
- Si la liste est vide alors premier et dernier valent NULL. Sinon, ils pointent respectivement sur le premier et le dernier maillon de la chaîne.

Question IV.22 Reprendre les questions précédentes avec ce nouveau type.

Démonstration.

Peu de modifications à faire.

```

liste liste_vide()
{
    liste l;
    l.premier = NULL;
    l.dernier = NULL;
    return l;
}

void liste_affiche(liste l)
{
    putchar('[');
    maillon *m = l.premier;
    while(m != NULL)
    {
        printf("%d", m->valeur);
        if(m->suivant != NULL)
            putchar(',');
        m = m->suivant;
    }
    putchar(']');
    putchar('\n');
}

maillon *maillon_creer(int valeur,
                      maillon *precedent, maillon *suivant)
{
    maillon *m = malloc(sizeof(maillon));
    m->valeur = valeur;
    m->suivant = suivant;
    m->precedent = precedent;
    return m;
}

void maillon_detruire(maillon *m)
{
    free(m);
}

```

```
void chaine_detruire(maillon *m)
{
    while(m != NULL)
    {
        maillon *suivant = m->suivant;
        maillon_detruire(m);
        m = suivant;
    }
}

void liste_detruire(liste l)
{
    chaine_detruire(l.premier);
}

int liste_longueur(liste l)
{
    int longueur = 0;
    maillon *m = l.premier;
    while(m != NULL)
    {
        longueur += 1;
        m = m->suivant;
    }
    return longueur;
}

int liste_tete(liste l)
{
    assert(l.premier != NULL);
    return l.premier->valeur;
}

liste liste_queue(liste l)
{
    assert(l.premier != NULL);
    liste q;
    q.premier = l.premier->suivant;
    q.dernier = l.dernier;
    return q;
}

void liste_ajout_en_tete(liste *l, int valeur)
{
    l->premier = maillon_creer(valeur, NULL, l->premier);
    if(l->dernier == NULL) // la liste était vide
        l->dernier = l->premier;
}

void liste_suppr_en_tete(liste *l)
{
    assert(l->premier != NULL);
    maillon *m = l->premier;
    l->premier = m->suivant;
    l->premier->precedent = NULL;
```

```

maillon_detruire(m); // on libère la mémoire
if(l->premier == NULL) // on a vidé la liste
    l->dernier = NULL;
}

void liste_ajout_en_fin(liste *l, int valeur)
{
    maillon *m = l->dernier;
    l->dernier = maillon_creer(valeur, m, NULL);
    m->suivant = l->dernier;
    if(l->premier == NULL) // la liste était vide
        l->premier = l->dernier;
}

maillon *liste_cherche_maillon(liste l, int valeur)
{
    maillon *m = l.premier;
    while(m != NULL && m->valeur != valeur)
    {
        m = m->suivant;
    }
    return m;
}

void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur)
{
    m->suivant = maillon_creer(valeur, m, m->suivant);
}

```

Question IV.23 Écrire une fonction de prototype

□ `void liste_suppr_en_fin(liste *l);`

qui supprime en $O(1)$ le dernier maillon.

Démonstration.

```

void liste_suppr_en_fin(liste *l);
{
    maillon *fin = l->dernier;
    l->dernier = fin->precedent;
    l->dernier->suivant = NULL;
    if (l->dernier == NULL)
        l->premier = NULL;
    maillon_detruire(fin);
}

```

Question IV.24 Écrire une fonction de prototype

□ `void liste_ajout_maillon_avant(liste *l, maillon *m, int valeur);`

qui ajoute un maillon avant m en $O(1)$.

Démonstration.

□ `void liste_ajout_maillon_avant(liste *l, maillon *m, int valeur)`

```
{  
    if (m == l->premier)  
        liste_ajout_en_tete(l, valeur);  
    else if (m == l->dernier)  
        liste_ajout_en_fin(l, valeur);  
    else  
    {  
        // En O(1)  
        m->precedent->suivant = maillon_creer(valeur,  
            m->precedent, m);  
        m->precedent = m->precedent->suivant;  
    }  
}
```

Question IV.25 Écrire une fonction de prototype

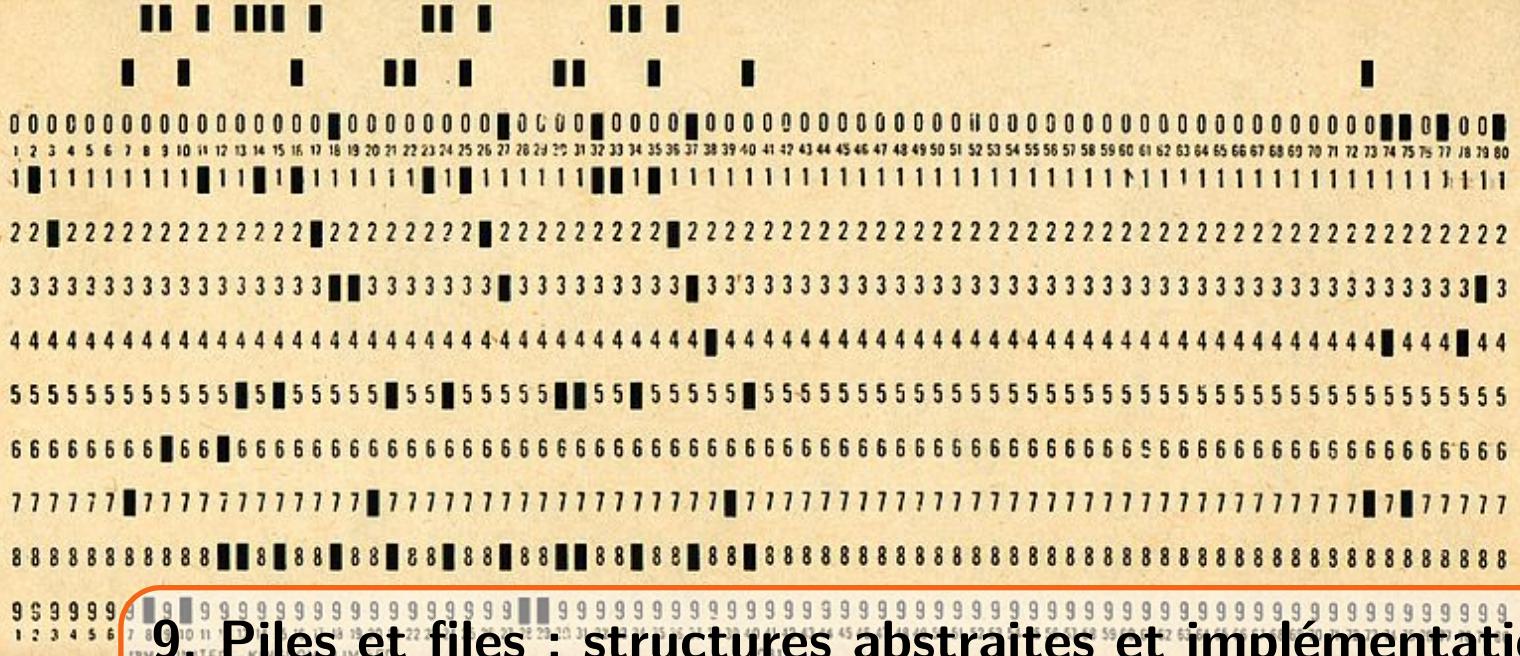
□ `void liste_suppr_maillon(liste *l, maillon *m);`

qui supprime en $O(1)$ le maillon m .

Démonstration.

□ `void liste_suppr_maillon(liste *l, maillon *m)`

```
{  
    if (m == l->premier)  
        liste_suppr_en_tete(l);  
    else  
    {  
        m->precedent->suivant = m->suivant;  
        m->suivant->precedent = m->precedent;  
        maillon_detruire(m);  
    }  
}
```

9. Piles et files : structures abstraites et implémentation

I Piles

1.1 Principe

Une pile est une structure de donnée abstraite permettant d'ajouter et de retirer des éléments selon le principe

LIFO : Last In First Out

C'est-à-dire que le premier élément retiré sera celui qui a été ajouté en dernier. La bonne manière de se représenter une pile est donc d'imaginer une pile de dossiers à traiter sur un bureau. Chaque nouveau dossier est empilé et on traite à chaque fois le dossier sur le dessus.

■ **Remarque 9.1** Ce n'est sûrement pas la manière la plus efficace de gérer des dossiers et l'on risque d'avoir des dossiers empilés depuis très longtemps sans être traités.

On va avoir quatre opérations :

- Créer une pile vide
- Tester si une pile est vide
- Ajouter un élément à une pile, on appelle cette opération *empiler* et en anglais *push*
- Retirer un élément à une pile et le renvoie, on appelle cette opération *dépiler* et en anglais *pop*

Le comportement opérationnel de cette structure abstraite est donné par le principe LIFO précédent.

On considère qu'il est très important que ces quatre opérations soient de complexité en temps constante, éventuellement amorti. Par contre, il est possible de limiter le nombre maximum d'éléments. C'est une vision qui est en accord avec le rôle des piles pour gérer des tâches à traiter pour lesquelles on a souvent un majorant sur leur nombre.

1.2 Représentation visuelle

Comme souvent, la représentation visuelle est importante car c'est elle qui permet de raisonner efficacement. Ici, on va représenter une pile verticalement ou horizontalement avec un côté fermé : le *fond* de pile et un côté ouvert permettant d'empiler et de dépiler des éléments.



1.3 Implémentations

1.3.i Dans un tableau borné

Reprenant la remarque précédente, on va présenter une implémentation très standard des piles dans des tableaux de taille fixe. L'idée est de considérer un tableau `t` de taille N muni d'un entier `fond` indiquant l'indice du prochain élément libre.

Au départ, `fond` vaut 0. Chaque fois qu'on empile un élément, on le place à l'indice indiqué par `fond` et on l'incrémente. Pour dépiler, on décrémente `fond` et on renvoie l'élément à cet indice. En fait, les éléments ne sont pas vraiment retirés. Il s'agit exactement du comportement de la pile d'exécution du compilateur vue au chapitre sec:pileexec.

Le choix de la valeur N donnant le nombre maximum d'éléments empilables est critique : il faut qu'elle soit grande devant l'estimation qu'on peut faire du nombre d'empilements. L'implémentation d'une telle pile se fait en général dans des conditions où l'on va éviter de vérifier l'absence de dépassement. Dans le code qui va suivre, on a fait le choix de passer par des `assert` pour vérifier ces conditions uniquement en mode développement.

■ **Remarque 9.2** En C, on passera le plus souvent par un tableau statique avec `N` défini à une grande valeur entière par un `#define`. On présente une version plus dans l'esprit de ces chapitres structures de données.

On va ainsi définir un type `pile` avec des pointeurs :

```
struct pile {
    int *t;
    int fond;
};

typedef struct pile pile;
```

Pour ajouter et retirer un élément, comme on va manipuler l'entier `fond`, il est nécessaire de passer `pile` par pointeur. On fait ainsi ce choix pour l'ensemble des fonctions.

L'opération de création doit faire une allocation et cela va donc nécessiter une opération explicite de destruction :

```
int N = 1000; // La pile sera de 1000 éléments au maximum

pile *pile_creer()
{
    pile *p = malloc(sizeof(pile));
    p->t = malloc(sizeof(int) * N);
    p->fond = 0;
    return p;
}

void pile_detruire(pile *p)
{
    free(p->t);
    free(p);
}
```

Empiler et dépiler correspond directement à la description précédente.

```
void empiler(pile *p, int x)
{
    assert(p->fond < N);
    p->t[p->fond] = x;
    p->fond++;
}

int retirer(pile *p)
{
    assert(p->fond > 0);
    p->fond--;
    return p->t[p->fond];
}

bool est_vide(pile *p)
{
    return p->fond == 0;
}
```

I.3.ii Dans un tableau dynamique

On a vu qu'on pouvait définir des tableaux redimensionnables et que cela permettait d'effectuer des ajouts et des suppressions en complexité amortie constante.

On peut donc directement réaliser une pile à l'aide de cette structure en ajoutant et supprimant des éléments. C'est l'interface usuelle avec le type `list` de Python qui propose déjà cela avec les fonctions `append` et `pop`. On peut rajouter une surcouche superficielle pour retrouver la nomenclature précédente :

```
def creer_pile():
    return []

def est_vide(p):
    return p == []
```

```
def empiler(p, x):
    p.append(x)

def depiler(p):
    return p.pop()
```

1.3.iii Avec des listes chaînées

Avec des listes chaînées, l'idée est d'ajouter et de supprimer des éléments en tête. En effet, on a vu que ces deux opérations étaient en temps constant, ce qui est particulièrement adapté aux piles.

En OCaml, on remarque que les listes par défaut conviennent, cependant, il est nécessaire de changer de liste pour rajouter ou supprimer un élément, c'est pourquoi on définit le type '`'a pile`' comme une '`'a list ref`' c'est-à-dire un pointeur sur un maillon de tête comme on aurait pu le faire en C.

On définit ainsi directement les quatre opérations :

```
type 'a pile = 'a list ref

let cree_pile () = ref []

let est_vide p = !p = []

let empile p x = p := x :: !p

let depile p =
    match !p with
    | [] -> failwith "Pile vide"
    | x::q -> p := q; x
```

OCaml

1.4 Applications

1.4.i Parenthésage

1.4.ii Évaluation d'expressions

!subsubsubsection(Expression postfixe) !subsubsubsection(Expression infixé)

II Files

II.1 Principe

Une file est une structure de donnée abstraite permettant d'ajouter et de retirer des éléments selon le principe

FIFO : First In First Out

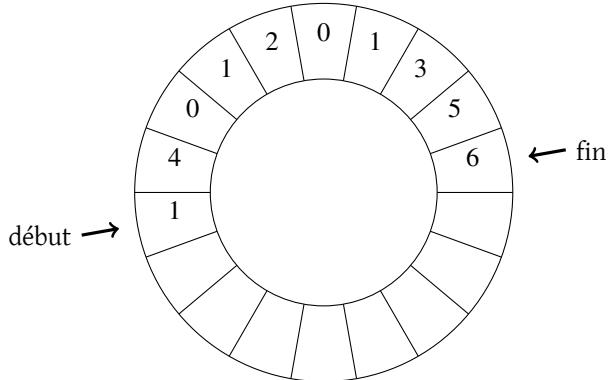
C'est-à-dire que le premier élément retiré sera celui qui a été ajouté le plus récemment. La bonne manière de se représenter une file est donc d'imaginer la file d'attente à la caisse d'un magasin.

II.2 Implémentations

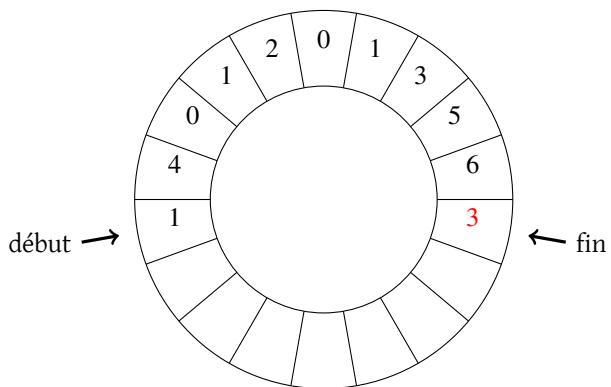
II.2.i Dans un tableau borné

On peut reprendre l'implémentation dans un tableau des piles. Ici, on va conserver deux indices : l'indice de fin et l'indice du début de la file.

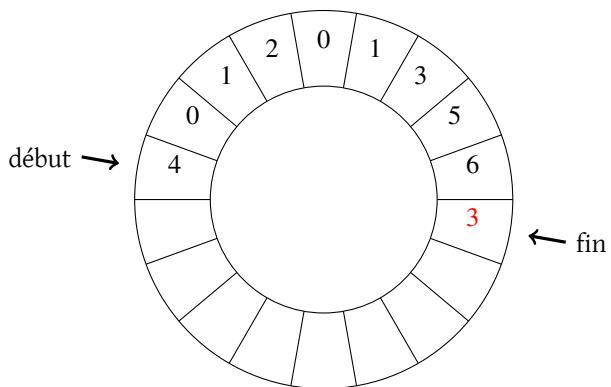
L'idée est de représenter une file comme une plage de cases dans un tableau circulaire :



Pour enfiler, il suffit de décaler fin d'une case vers la droite et d'ajouter à cette case l'élément. Ainsi si on enfile 3 dans l'exemple précédent on obtient :



Pour défiler, on va renvoyer l'élément pointé par début et décaler celui-ci d'une case vers la droite. Dans l'exemple précédent, on obtient alors la valeur 1 et le tableau devient :



Si, comme pour les piles, on suppose que la taille de la file ne dépassera jamais le nombre de cases du tableau circulaire, il n'y a pas de risques de dépassement. La question qui se pose alors est celle de l'implémentation d'un tableau circulaire de N cases dans un tableau usuel de N cases.

La technique consiste à couper le tableau arbitrairement entre deux cases et à l'aplatir :



Maintenant, la question du bord se pose : les décalages doivent se faire modulo la longueur du tableau pour avoir le même comportement qu'un tableau circulaire.

On traduit ensuite directement cette représentation en C :

```

struct file {
    int *t;
    int debut;
    int fin;
};

typedef struct file file;

```

II.2.ii Avec des listes chaînées

II.2.iii Avec deux piles

II.3 Applications



10. Arbres

Arbres binaires

1.1 Définition inductive

Définition I.1 Un arbre binaire étiqueté par \mathcal{E} est :

- soit vide, et on le note alors nil ou \perp
- soit un triplet (g, x, d) où $x \in \mathcal{E}$ et g et d sont des arbres binaires.

■ **Remarque 10.1** Cette définition inductive ressemble à la définition des listes.

On peut la formaliser en introduisant $T_b(\mathcal{E})$ l'ensemble des arbres étiquetés par \mathcal{E} . C'est le **plus petit ensemble** tel que :

- $\text{nil} \in T_b(\mathcal{E})$
- $\forall x \in \mathcal{E}, \forall g, d \in T_b(\mathcal{E}), (g, x, d) \in T_b(\mathcal{E})$

Les conséquences de la précision **plus petit ensemble** sont importantes :

- les arbres sont nécessairement des expressions finies, c'est-à-dire qu'il ne comportent qu'un nombre fini de constructions. C'est automatique car l'ensemble des arbres finis vérifie les conditions précédentes ;
- pour tout triplet (g, x, d) il ne peut exister qu'un arbre, sinon, en enlevant un arbre en double on vérifierait encore les deux conditions précédentes.

On en déduit directement la notion de preuve par **induction structurelle** sur les arbres :

$$\forall a \in T_b(\mathcal{E}), P(a) \iff \begin{cases} P(\text{nil}) \\ \forall x \in \mathcal{E}, \forall g, d \in T_b(\mathcal{E}), P(g) \wedge P(d) \Rightarrow P(g, x, d) \end{cases}$$

Cette induction correspond à un pseudo principe de récurrence qui nous permettra de faire les preuves.

Pour démontrer cette inégalité, il suffit de vérifier que $T_P = \{t \in T_b(\mathcal{E}), P(t)\}$ vérifie les relations de la définition inductive et $T_P \subset T_b(\mathcal{E})$ et par minimalité, on a bien $T_P = T_b(\mathcal{E})$.

■ **Exemple 10.1** Si $\mathcal{E} = \mathbb{N}$, les éléments suivants sont des arbres :

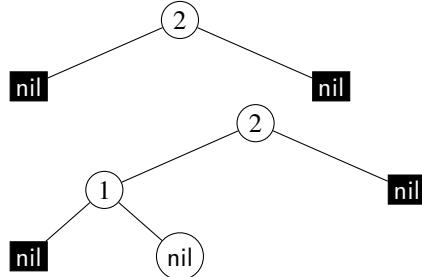
- nil
- (nil, 2, nil)
- ((nil, 1, nil), 2, nil)

On adoptera une représentation graphique très naturelle pour les arbres binaires où un

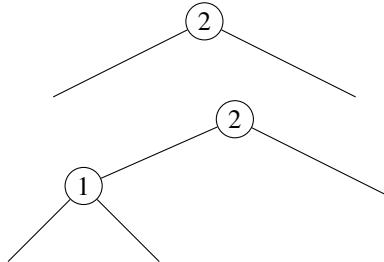


■ **Exemple 10.2** Les trois arbres précédentes sont donc représentés par :

nil



Afin d'alléger la notation, on omettra nil, sauf pour l'arbre vide. On fera cependant attention à conserver les arêtes donnant sur nil pour ne pas confondre $(\text{nil}, 1, \text{nil}(\text{nil}, 2, \text{nil}))$ et $((\text{nil}, 2, \text{nil}), 1, \text{nil})$. Ainsi on représentera les arbres précédents par :



1.2 Vocabulaire

Définition 1.2 Soit a un arbre binaire. Les arbres non vides présents dans a sont appelés les **nœuds** de a . Parmi ceux-ci on distingue ceux qui sont de la forme $(\text{nil}, x, \text{nil})$, appelés des **feuilles**. Les nœuds qui ne sont pas des feuilles sont appelés des **nœuds internes**. Le nœud a lui-même est appelé la racine de l'arbre. On note $N(a)$ les nœuds de a .

Si $n \in N(a)$ n'est pas la racine, il est le fils gauche ou le fils droit d'un unique nœud appelé le **père** de x .

Si $n \in N(a)$, on appelle **sous-arbre** de a l'arbre dont n est la racine.

■ **Remarque 10.2** En partant de la définition inductive, il y a une identification entre un sous-arbre et sa racine. Mais afin de raisonner, on distinguera un nœud en tant qu'emplacement au sein d'un arbre et le sous-arbre lui-même.

Définition 1.3 Soit a un arbre binaire.

- On appelle **taille** de a , et on note $|a|$ le nombre de nœuds de a .

- On appelle **hauteur** de a l'entier

$$h(a) = \begin{cases} -1 & \text{si } a = \text{nil} \\ 1 + \max(h(g), h(d)) & \text{si } a = (g, x, d) \end{cases}$$

Définition I.4 Soit a un arbre binaire et $n \in N(a)$. On appelle **profondeur** de n l'unique entier $p(a)$ tel qu'il existe une suite finie $(n_0, n_1, \dots, n_{p(a)})$ de n vérifiant :

- n_0 est la racine de a
- $n_{p(a)} = n$
- pour tout i , n_i est le père de n_{i+1}

Cette suite finie est le **chemin** de la racine à n . Il est nécessairement unique car chaque nœud autre que la racine a un unique père.

Théorème I.1 Soit a un arbre, si a est non vide, alors $h(a) = \max_{n \in N(a)} p(n)$.

Démonstration.

Par induction structurelle sur a .

- **Initialisation** Si $a = \text{nil}$ la prémissse est fausse, donc l'implication est trivialement vérifiée.
- **Hérédité** Supposons la propriété vérifiée pour deux arbres g et d , soit $x \in \mathcal{E}$, on va montrer qu'elle est vérifiée pour $a = (g, x, d)$. On a quatre cas pour le couple (g, d) :

- ★ Soit $g \neq \text{nil}$ et $d \neq \text{nil}$. Dans ce cas, par hypothèse $h(g) = \max_{n \in N(g)} p_g(n)$ où p_g est la profondeur de n en tant que nœud de l'arbre g . Or, mis à part la racine de a , le chemin menant dans a au nœud n est dans g . On a donc directement $p_g(n) = p(n) - 1$. Ainsi $h(g) = \max_{n \in N(g)} p(n) - 1$. De même, $h(d) = \max_{n \in N(d)} p(n) - 1$. On a

$$h(a) = 1 + \max(h(g), h(d)) = \max(\max_{n \in N(g)} p(n), \max_{n \in N(d)} p(n))$$

Or, le seul nœud de a qui n'est ni dans g ni dans d est sa racine, qui est de profondeur nulle donc $h(a) = \max_{n \in N(a)} p(n)$.

- ★ Soit $g = \text{nil}$ et $d \neq \text{nil}$. Ainsi $h(g) = -1$ et donc $h(a) = 1 + h(d) = \max_{n \in N(d)} p(n)$ par l'analyse précédente. On conclut donc avec la propriété voulue.
- ★ Soit $g \neq \text{nil}$ et $d = \text{nil}$. Cas symétrique du précédent.
- ★ Soit $g = d = \text{nil}$. Auquel cas, $h(a) = 0$ qui est bien la profondeur de son unique nœud.

On a bien montré la propriété voulue par induction structurelle. ■

I.3 Implémentations

I.3.i En OCaml

En OCaml, on traduit directement la définition inductive par un type récursif :

```
OCaml | type 'a arbre = Noeud of 'a arbre * 'a * 'a arbre | Nil
```

On pourra alors définir des fonctions récursives sur les arbres par induction structurelle à l'aide d'un filtrage.

```
OCaml  let rec taille a = match a with
| Nil -> 0
| Noeud(g,x,d) -> 1 + taille g + taille d

let rec hauteur a = match a with
| Nil -> -1
| Noeud(g, x, d) -> 1 + max (hauteur g) (hauteur d)
```

I.3.ii En C

I.4 Arbres binaires stricts

Définition I.5 Un arbre binaire est dit **strict** lorsqu'aucun de ses nœuds n'a qu'un fils vide.

Cela revient à dire que les seules les feuilles ont des fils vides. On peut donc omettre l'arbre vide et ne garder que les feuilles à la place.

■ **Remarque 10.3** L'arbre de gauche ici est un arbre binaire strict alors que ce n'est pas le cas de l'arbre de droite :



Théorème I.2 Soit a un arbre binaire strict non vide. Si a a n nœuds internes, alors il a $n + 1$ feuilles.

Démonstration.

Par induction structurelle :

- **Initialisation** Si a a 0 nœud interne, c'est une feuille et on a directement la relation.
- **Hérédité** Supposons que la propriété soit vraie pour des arbres g et d , et soit $a = (g, x, d)$ un arbre dont ce sont les fils. On note $n_i(t)$ le nombre de nœuds internes et $n_f(t)$ le nombre de feuilles de l'arbre t . On a $n_i(a) = 1 + n_i(g) + n_i(d)$ et $n_f(a) = n_f(g) + n_f(d) = 2 + n_i(g) + n_i(d) = 1 + n_i(a)$. La propriété est démontrée pour a .

Cela permet naturellement de considérer des arbres où les feuilles et les nœuds internes ont deux types différents d'étiquettes.

L'exemple classique d'un tel arbre est celui des expressions arithmétiques :

- les nœuds internes sont étiquetés par des opérateurs binaires
- les feuilles sont étiquetées par des nombres.

En OCaml, on peut ainsi représenter un tel arbre par le type récursif :

```
OCaml  type ('a, 'b) arbre_bin = Feuille of 'a
      | Noeud of ('a, 'b) arbre_bin * 'b * ('a, 'b) arbre_bin
```

■ **Remarque 10.4** On ne peut plus représenter l'arbre vide avec ce type.

Et pour les expressions, on pourra ainsi définir un type pour les opérateurs et écrire :

```
Ocaml type operator = Plus | Times | Minus | Divides
let e = Noeud(Feuille 2, Plus, Noeud(Feuille 3, Mult, Feuille 4))
```

L'évaluation de telles expressions est étudiée dans le TP si dessous.

1.5 Arbres binaires complets

Définition 1.6 Un arbre binaire dont tous les niveaux sont plein sauf éventuellement le dernier est dit **complet**. Si son dernier niveau est également plein, on dit qu'il est **parfait**.

On peut représenter un arbre complet dans un tableau niveau par niveau en partant de la racine.

Si on place les indices comme étiquettes on aura, par exemple :

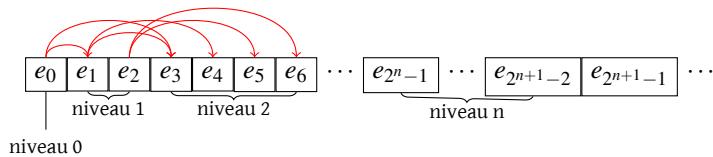


Et on pourra donc le représenter par le tableau des étiquettes dans cet ordre d'énumération. On remarque que :

- le nœud 0 est toujours la racine
- si on considère un nœud i , son fils gauche est le nœud $2i + 1$ et son fils droit est le nœud $2i + 2$.
- le fils gauche de $i + 1$ est $2(i + 1) + 1 = 2i + 3 > 2i + 2$. On est bien directement après.
- le k ème nœud du niveau l , en commençant à numérotier à 0, a pour indices $\sum_{j=0}^{l-1} 2^j + k = 2^l - 1 + k$.
- si $i > 0$, son père est le nœud $\lfloor \frac{i-1}{2} \rfloor$.

On peut donc, avec cette représentation, manipuler assez simplement un arbre binaire à **plat**

Si le nœud de numéro i a l'étiquette e_i , le tableau a la structure suivante :



II Arbres

II.1 Définition

On étend directement la définition inductive des arbres binaires au cas où les nœuds plus d'éléments.

Définition II.1 Un arbre étiqueté par \mathcal{E} est un couple (x, f) où $x \in \mathcal{E}$ et f une suite **finie** d'arbres.

On étend naturellement le vocabulaire des arbres binaires :

- le nœud de tête d'un arbre, en fin de compte sa *valeur*, est appelé sa **racine**
 - les éléments de f sont appelés les **fils** du nœuds
 - si f est vide, on dit que le nœud est une **feuille**, sinon, on dit que c'est un **nœud interne**.
 - $|f|$, le nombre d'éléments de f , est appelé l'**arité** du nœud. Les feuilles sont donc les nœuds zéroaire.
 - une suite finie d'arbres, ou plus généralement un ensemble d'arbres, est appelé une **forêt**.
- Si on note $T(\mathcal{E})$ l'ensemble des arbres étiquetés par \mathcal{E} , on a

$$\forall x \in \mathcal{E}, \forall n \in \mathbb{N}, \forall a_1, \dots, a_n \in T(\mathcal{E}), (x, (a_1, \dots, a_n)) \in T(\mathcal{E})$$

Attention contrairement à ce que peut laisser entendre la formule précédente, il est tout à fait possible que $n = 0$ ce qui correspond à la suite finie vide et donc à une feuille. On perd ici la possibilité de représenter un arbre vide.

■ Remarque 10.5 Un arbre binaire est un arbre dont les nœuds sont soit unaires soit binaires. On remarque en disant cela qu'on perd une information : la position gauche ou droite de l'unique fils d'un nœud unaire.

Comme pour les arbres binaires stricts, on pourra être amenés à considérer des arbres dont les nœuds internes et les feuilles sont étiquetés par des arbres

■

II.2 Implémentation

L'implémentation d'un arbre repose sur l'implémentation d'une suite finie. Celle-ci peut être faite par un tableau ou par une liste chaînée.

On retrouve alors des implementations différentes des arbres mais qui sont, au fond, très proches.

En OCaml, on pourra alors avoir les deux implémentations suivantes :

```
OCaml | type 'a arbre_l = { etiquette : 'a; enfants : 'a arbre_l list }
      | type 'a arbre_a = { etiquette : 'a; enfants : 'a arbre_a array }
```

On va alors avoir des programmes à la présentation assez différente suivant le type choisi. Voici, par exemple, les deux implementations du calcul de la taille d'un arbre.

Avec des listes et aucune fonction du module `List`, on écrit souvent une fonction récursive sur les arbres et une fonction récursive sur les forêts.

```
OCaml | let rec taille_arbre a =
      1 + taille_foret a.enfants
      and taille_foret l = match l with
        | [] -> 0
        | t::q -> taille_arbre t + taille_foret q
```

On peut aussi utiliser directement `fold_left` et `map` :

```
OCaml | let rec taille_arbre a =
      1 + List.fold_left (+) 0 (List.map taille_arbre a.enfants)
```

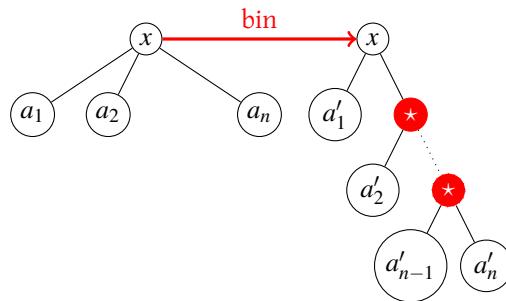
Avec l'autre représentation, on procédera avec un mélange de fonction récursive et de boucles. Cela n'est pas sans rappeler ce qui a été fait pour le backtracking.

```
Ocaml
let rec taille_arbre a =
  let t = ref 1 in
  for i = 0 to Array.length a.enfants - 1 do
    t := !t + taille_arbre a.enfants.(i)
  done;
  !t
```

II.3 Représentation par un arbre binaire

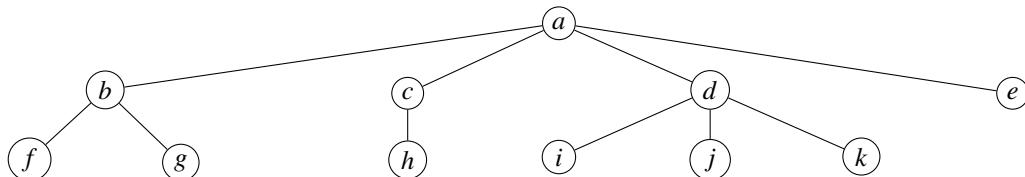
Pour des raisons d'efficacité ou de réutilisation de programmes existants, on peut vouloir représenter un arbre comme un arbre binaire.

On va définir une application : $\text{bin} : T(\mathcal{E}) \rightarrow T_b(\mathcal{E} \cup \{\star\})$ ainsi : $\text{bin}(x, ()) = (\text{nil}, x, \text{nil})$, $\text{bin}(x, (a)) = (\text{bin}(a), x, \text{nil})$ et pour un nœud d'arité strictement plus grande que 1 :

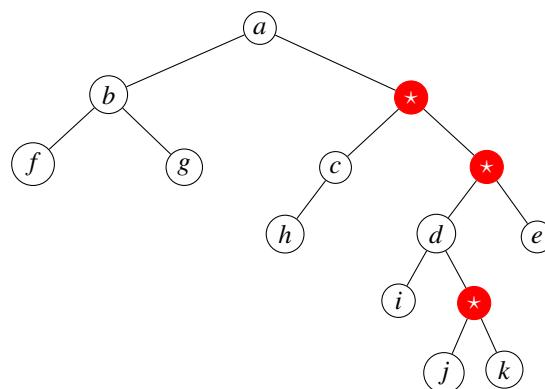


où $a'_i = \text{bin}(a_i)$. Les nœuds rouges sont des nœuds ayant une étiquette spéciale \star telle que $\star \notin \mathcal{E}$.

■ **Exemple 10.3** L'arbre suivant :



sera alors représenté par l'arbre binaire :



II.4 Applications

II.4.i Arbres d'expressions

On a déjà vu l'utilisation des arbres binaires pour représenter des expressions arithmétiques avec des opérateurs unaires ou binaires. On généralise naturellement cela à des expressions dont les opérateurs sont d'arité quelconque.

C'est ainsi qu'on représente un programme en mémoire, on parle d'arbre syntaxique.

II.4.ii Arbres préfixes ou tries

Un *arbre préfixe* ou *trie* est un arbre permettant de représenter un ensemble de mots. On étiquette les arêtes par des caractères et les nœuds par un booléen indiquant si la suite des étiquettes qui mène de la racine à ce nœud est un mot.

Par exemple, le trie :



permet de représenter l'ensemble de mots : bar, barbe, art, ami, abri, abricot, abonne.

Afin de représenter un tel arbre, il est nécessaire d'avoir une étiquette sur les arêtes, le plus simple pour cela est de remplacer la liste des enfants par une liste de couples (étiquette, enfant) ainsi :

```
OCaml | type trie = {
    mot : bool;
    enfants : (char * trie) list
}
```

Les fonctions de manipulations de ce type de donnée sont étudiées dans le TP TODO.

III Parcours

III.1 Parcours récursif d'un arbre binaire

On a déjà vu dans les paragraphes précédent ce qui constitue le cœur du parcours d'un arbre :

```
OCaml | let rec parcours a =
    match a with
    | Nil -> ()
    | Noeud(g, x, d) ->
        parcours g;
```

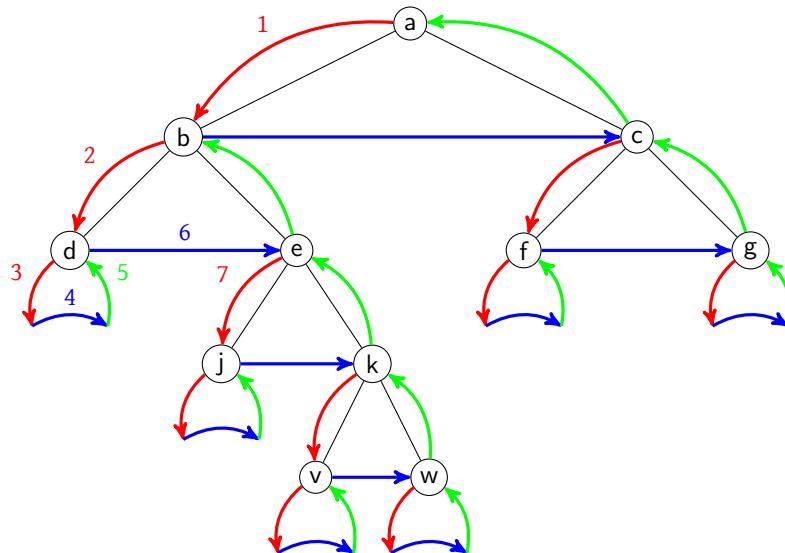
parcours d

Cette fonction ne fait rien, mais elle va parcourir chacun des nœuds de l'arbre. Si on veut effectuer un traitement sur le nœud, on peut le faire à trois moments :



- Avant l'appel à `parcours g`, on parle de *traitement préfixe*
- Entre les deux appels, on parle de *traitement infixé*
- Après l'appel à `parcours g`, on parle de *traitement postfixé*

On représente ici l'ordre dans lequel on va effectuer chacun de ces trois traitements selon le code couleur : rouge pour préfixe, bleu pour infixé et vert pour postfixé. On a également indiqué, pour les premiers traitements, l'ordre dans lequel ils sont effectués par des numéros.



Pour illustrer ces traitements, on peut rajouter trois arguments au parcours :

```

let rec parcours prefixe infixé postfixé a =
  match a with
  | Nil -> ()
  | Noeud(g, x, d) ->
      prefixe a;
      parcours g;
      infixé a;
      parcours d;
      postfixé a
  
```

OCaml

On peut, par exemple, définir les deux fonctions suivante :

```

let idle a = () (* ne fait rien *)
let print a = match a with
  | Nil -> failwith "Vide"
  
```

OCaml

| Noeud(_,x,_) -> print_char a

On peut alors observer les noeuds affichés sur l'arbre précédent selon les traitements effectués :

- parcours print idle idle a : abdejkvwcfg on suit uniquement les arêtes rouges
- parcours idle print idle a : dbejvkwafcg on suit uniquement les arêtes bleues
- parcours idle idle print a : djvwkebfgca on suit uniquement les arêtes vertes
- parcours print print print a : abdddbejjjekvvvkwwwkeb... on suit toutes les arêtes

■ **Remarque 10.6** Si on considère un arbre d'expression comme :



On va alors obtenir en affichant à chaque traitement la suite : +222 + ×333 × 444 × +. Chaque nombre n étant une feuille, on va observer l'affichage de nnn qu'on peut réduire à n . Pour les opérateurs, si on remplace le premier affichage par (et le dernier par). En procédant ainsi, on affiche $(2 + (3 \times 4))$ et on ainsi retrouvé l'écriture bien parenthésée de l'expression.

Définition III.1 L'ordre dans lequel on effectue un traitement *préfixe* dans un parcours sur les noeuds d'un arbre est appelé l'ordre *préfixe* sur les noeuds.

On définit, de même, l'ordre *infixe* et l'ordre *postfixe*.

III.2 Parcours impératifs

III.2.i Cadre général

On va considérer une structure de donnée abstraite qui généralise les piles et les files et permettre de représenter un ensemble de tâches à traiter. Pour cela, on dispose d'un type paramétrique 'a t et de l'interface :

- cree : unit -> 'a t crée un ensemble de tâches
- ajoute : 'a t -> 'a -> unit ajoute une nouvelle tâche à traiter
- retire : 'a t -> 'a retire une tâche de l'ensemble des tâches à traiter
- est_vide : 'a t -> bool renvoie un booléen indiquant si l'ensemble de tâches est vide

On a ainsi vu deux possibles implémentations, qui sont elles-mêmes des structures abstraites mais un peu moins abstraites que celle-ci :

- les piles pour lesquelles on retire le dernier élément ajouté
- les files pour lesquelles on retire l'élément le plus anciennement ajouté

On écrire un parcours impératif générique :

```

let parcours traitement a =
  let avisiter = cree () in
  ajoute avisiter a;
  while not (est_vide avisiter) do
    let a = retire avisiter in
    match a with
      | Nil -> ()
  
```

Ocaml

```

| Noeud(g, _, d) -> traitement a;
| ajoute avisiter g;
| ajoute avisiter d
done

```

III.2.ii Cas du parcours en profondeur

Dans le cas où on utilise une pile, on retrouve essentiellement le parcours récursif précédent avec quelques changements :

- on empile ici d'abord g puis d, donc on va à droite avant d'aller à gauche. Il suffit de permuter les deux ajouts pour retrouver l'ordre précédent
- on effectue un unique traitement préfixe
- si on considère la pile d'appels récursifs du parcours récursif, on constate que le nombre de structure pile est majoré par la hauteur de l'arbre. Ici, on va potentiellement ajouter tous les nœuds sur la pile, donc, on a une complexité en espace en $O(|a|)$ plutôt qu'en $O(h(a))$.

On parle de **parcours en profondeur** ou depth-first search (DFS) en anglais.

■ **Exemple 10.4** Si on considère l'arbre suivant :



En ignorant les arbres vides, par exemple en ne les ajoutant pas, on va avoir le déroulement suivant du parcours :

avisiter	étiquette traitée
1 →	
2, 3 →	1
2, 5, 6 →	3
2, 5 →	6
2, 7 →	5
2 →	7
4 →	2
	4

III.2.iii Cas du parcours en largeur

Dans le cas où on utilise une file, on obtient un parcours appelé le **parcours en largeur** en breadth-first search (BFS) en anglais.

Dans ce parcours, on va visiter les nœuds *niveau par niveau* en partant de la racine jusqu'aux feuilles de plus grande profondeur.

■ **Exemple 10.5** Si on considère l'arbre suivant :



En ignorant les arbres vides, par exemple en ne les ajoutant pas, on va avoir le déroulement suivant du parcours :

aviser	étiquette traitée
→ 1 →	
→ 3, 2 →	1
→ 4, 3 →	2
→ 6, 5, 4 →	3
→ 6, 5 →	4
→ 7, 6 →	5
→ 7 →	6
	7

Ce parcours est particulièrement intéressant quand on cherche une information avec la plus petite profondeur possible. Par exemple, dans le cas de la recherche d'une solution à un problème, on peut vouloir tester les petites solutions avant les plus grandes.

Pour des problèmes d'énumération, c'est également intéressant, car on va obtenir des éléments par ordre croissant de la longueur du chemin depuis la racine. Par exemple, pour les tries, on en déduit les mots par ordre croissant de longueur.

III.3 Parcours d'arbres

Pour des arbres, on va suivre les mêmes principes. La différence va se situer au niveau de l'implémentation car il faudra alors manipuler des listes ou des tableaux d'enfants.

Notons qu'on a déjà vu un tel problème quand on a résolu des problèmes par backtracking. En effet, avec le backtracking, on a un arbre implicite des positions partielles dont les enfants sont les mouvements possibles vers de nouvelles positions.

IV Arbres binaires de recherche

IV.1 Objectif

On souhaite ici réaliser une structure efficace d'ensembles finis. Pour cela, on cherche à définir une structure immuable 'a ensemble muni de quatre opérations

- `ensemble_vide` : 'a ensemble l'ensemble vide
- `ajoute` : 'a ensemble -> 'a -> 'a ensemble rajoute un élément à un ensemble et renvoie le nouvel ensemble
- `supprime` : 'a ensemble -> 'a -> 'a ensemble retire un élément à un ensemble et renvoie le nouvel ensemble
- `contient` : 'a ensemble -> 'a -> bool teste si l'ensemble contient un élément
- `cardinal` : 'a ensemble -> int renvoie le nombre d'éléments de l'ensemble

Notons qu'on pourra souvent relâcher la contrainte naturelle des ensembles en permettant d'ajouter plusieurs fois un même élément.

Une implémentation possible de cette structure serait d'utiliser des listes ou des tableaux triés, on aurait alors des opérations en $O(n)$ pour manipuler n éléments ($O(\log_2 n)$ en optimisant la recherche avec une recherche dichotomique).

Ici, on présente une implémentation rendant possible une complexité en $O(\log_2 n)$ pour chaque opération sous certaines hypothèses dont on montrera qu'elles peuvent être satisfaites dans un second temps.

IV.2 Définition

Définition IV.1 On dit qu'un arbre a étiquetted par X , muni d'une relation d'ordre total, est un **arbre binaire de recherche** (abr) lorsque :

- soit $a = \text{nil}$
- soit $a = (g, x, d)$ où g et d sont des arbres binaires de recherches et de plus

$$\max_{y \in g} e(y) \leq x \leq \min_{y \in d} e(y)$$

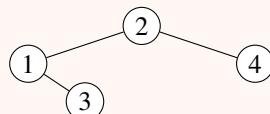
où $e(y)$ est l'étiquette du nœud y .

- **Remarque 10.7**
 - Tout sous-arbre d'un arbre binaire de recherche est un arbre binaire de recherche.
 - Pour un ensemble donné d'étiquettes il n'y a pas unicité de l'arbre binaire de recherche : voir ci-après.

Exemples d'abr :



■ **Remarque 10.8** La propriété caractéristique des ABR n'est pas locale c'est-à-dire tester si chaque nœud a une étiquette supérieure à son fils gauche et inférieure à son fils droit ne suffit pas.



IV.3 Opérations

IV.3.i Minimum et maximum

On peut obtenir la valeur minimale et maximale d'un abr en effectuant un parcours le long de la branche la plus à gauche ou de la branche la plus à droite.

En effet, si l'arbre est (g, x, d) : soit g est vide et $x \leq \min d$ est le minimum, soit g est non vide et on a alors $\min g \leq \max g \leq x$ donc le minimum de g est celui de l'arbre.

On en déduit le programme suivant :

```
Ocaml let rec minimum a =
  match a with
  | Nil -> failwith "vide"
  | Noeud(Nil, x, _) -> x
  | Noeud(g, _, _) -> minimum g
```

On remarque que pour parcourir cette branche, on est en $O(h(a))$ où a est l'arbre considéré.

!susubsection(Test si un arbre est un abr) Pour tester si un arbre est bien un arbre binaire de recherche on doit connaître la valeur maximum de l'arbre gauche et minimum de l'arbre droit et tester si l'étiquette de chaque nœud n vérifie

$$\max_{\eta \in A_g} e(\eta) \leq e(n) \leq \min_{\eta \in A_d} e(\eta)$$

On peut écrire une procédure qui indique si l'arbre est un arbre binaire de recherche et les valeurs maximum et minimum de l'arbre (puisque'on doit les calculer...)

```
Ocaml let rec abr_test a = match a with
  | Nil -> failwith "Arbre vide"
  | Noeud(Nil,n,Nil) -> (true, n, n)
  | Noeud(Nil,n,d) -> let bd, md, Md = abr_test d in
    (bd && n < md, min n md, max n Md)
  | Noeud(g,n,Nil) -> let bg, mg, Mg = abr_test g in
    (bg && n >= Mg, min n mg, max n Mg)
  | Noeud(g,n,d) ->
    let bg, mg, Mg = abr_test g in
    let bd, md, Md = abr_test d in
    (bg && n >= Mg && n < md, min n (min mg md), max n (max Mg Md))
```

Ici on a fait le choix de ne rien associer à l'arbre vide et de le traiter en amont au niveau des nœuds. Cela permet de ne pas imposer de contraintes sur le type des étiquettes comme cela aurait été le cas en mettant des valeurs ad hoc dans le cas de l'arbre vide (typiquement $\pm\infty$).

■ **Remarque 10.9** On rajoute des fonctions permettant de manipuler facilement ces données et on obtient alors un algorithme de test plus concis :

```

OCaml
let valeur a = match a with
| None -> failwith "None n'a pas de valeur"
| Some x -> x

let opt_bin f a b = match a, b with
| None, _ -> b
| _, None -> a
| Some a, Some b -> Some (f a b)

let omax = opt_bin max
let omin = opt_bin min

let abr_test a =
  let rec aux a = match a with
    | Nil -> (true, None, None)
    | Noeud(g,n,d) ->
        let bg, mg, Mg = aux g in
        let bd, md, Md = aux d in
        (bg && (Mg = None || n >= valeur Mg)
         && (md = None || n < valeur md),
          omin (Some n) (omin mg md),
          omax (Some n) (omax Mg Md))
  in let est_abr, min_abr, max_abr = aux a in
  (est_abr, valeur min_abr, valeur max_abr)

```

On verra en exercice qu'une lecture infixe permet de conclure également.

IV.3.ii Recherche d'un élément

Comme on leur nom l'indique, la recherche est adaptée à ce type d'arbres. Le modèle par adjonction s'applique et donc la hauteur moyenne d'un arbre binaire de recherche est un $\mathcal{O}(\log_2(n))$. On peut construire la fonction de recherche :

```

OCaml
let rec recherche x a = match a with
| Nil -> false
| Noeud(gauche,n,droite) when x=n -> true
| Noeud(gauche,n,droite) when x>n -> recherche x droite
| Noeud(gauche,n,droite) -> recherche x gauche

```

IV.3.iii Insertion d'un élément

Étant donné un arbre binaire de recherche a et une étiquette x , on veut construire un nouvel arbre binaire de recherche contenant les étiquettes des noeuds de a et x . Il n'y a bien sur pas unicité de la solution... Dans un arbre binaire de recherche, l'opération se décompose en une étape de recherche, qui renvoie des informations sur la place où doit être inséré le nouvel élément, suivie de l'adjonction proprement dite.

```

OCaml
let rec insere x a = match a with
| Nil -> Noeud(Nil,x,Nil)
| Noeud(gauche,n,droite) when x>n -> N(gauche,n,insere x droite)
| Noeud(gauche,n,droite) -> N(insere x gauche,n,droite)

```

On compare x à la racine pour déterminer s'il faut l'ajouter dans le sous-arbre gauche ou le sous-arbre droit, et l'on rappelle la procédure récursivement. Le dernier appel récursif se fait sur un arbre vide et l'on crée alors à cette place le noeud d'étiquette x . Le nouvel arbre est recons-

ruit de proche en proche depuis cette branche vers la racine, sans modifier les branches non explorées. Les anciens nœuds ne sont pas modifiés et l'ancien arbre reste donc disponible.

Construction par insertion successive de l'ABR pour les éléments 5, 9, 1, 20, 4, 7, 17, 12 :



IV.3.iv Suppression d'un élément

Le problème de la suppression d'un élément (ou plutôt de sa première occurrence) d'un arbre binaire de recherche est plus compliqué. On peut le décomposer en :

- Recherche de l'élément
- Suppression
- Recomposition de l'arbre

Le nœud à supprimer est la racine d'un sous-arbre de l'arbre binaire de recherche. Commençons par considérer le problème de la suppression de la racine z d'un arbre binaire de recherche.

- Si les deux branches issues de z sont vides, on supprime et c'est fini (cas (a) de la figure $z = 13$).
- Si l'une des deux branches issues z est vide, on se contente de garder l'autre branche. On conserve ainsi un arbre binaire de recherche (cas (b) de la figure $z = 16$).
- Si les deux branches sont non vides, soit y l'élément le plus à gauche du sous-arbre droit A_d de z . On a $y > z$ par la propriété des ABR.

Remarquons que y ne peut avoir de fils gauche car sinon il ne serait pas le plus à gauche de A_d .

On peut donc le supprimer et le reporter en lieu et place de z : on obtient alors un nouvel arbre qui est un arbre binaire de recherche. En effet :

- La branche droite est un arbre binaire de recherche (on a supprimé un nœud sans fils gauche) et l'étiquette de la racine est, par construction, inférieure aux étiquettes des nœuds de cette branche.
- La branche gauche est un arbre binaire de recherche (on n'y a pas touché) et la nouvelle étiquette de la racine y est supérieure ou égale à z qui majorait les étiquettes des nœuds de la branche gauche.

(cas (c) de la figure $z = 5$ et $y = 6$). La localisation de y est facile : c'est nécessairement l'élément le plus à gauche de A_d .

Les trois cas de suppression :



On en déduit le programme suivant :

```

let rec supprime a x =
  match a with
  | Nil -> Nil
  | Noeud(g, y, d) when y < x ->
    Noeud(g, y, supprime d x)
  | Noeud(g, y, d) when y > x ->
    Noeud(supprime g x, y, d)
  (* à partir d'ici l'étiquette est forcément x *)
  | Noeud(nil, _, nil) -> nil
  | Noeud(nil, _, d) -> d
  | Noeud(g, _, nil) -> g
  | Noeud(g, x, d) ->
    let y = minimum d in
    Noeud(g, y, supprime d y)
  
```

Ocaml

IV.4 Équilibrage

IV.4.i Principe et approche naïve

Les opérations précédentes sont toutes en $O(h(a))$, or on sait que $h(a) \leq |a| \leq 2^{h(a)} - 1$ donc $\log_2(|a| + 1) \leq h(a) \leq |a|$. On aimerait se rapprocher au plus de $\log_2 |a|$ et pour cela, il faut que l'arbre soit le plus proche possible d'un arbre parfait. On parle alors d'équilibrage.

Une approche naïve consiste à extraire les valeurs dans l'ordre croissant à l'aide d'un parcours infixé puis à reconstruire un arbre équilibré par dichotomie : on place la valeur médiane à la racine et on construit récursivement les sous-arbres gauche et droite.

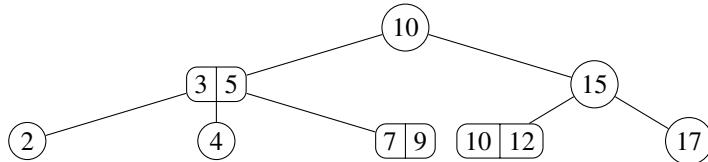
IV.4.ii Arbres 2-3

Définition IV.2 Un arbre 2-3 est un arbre étiqueté contenant des nœuds de deux types :

- des nœuds binaires $N(g, x, d)$ où g, d sont des arbres et x une étiquette tel que $\max g \leq x \leq \min d$ (c'est la même condition que les ABR) ;
 - des nœuds ternaires $M(g, x, m, y, d)$ où g, m, d sont des arbres 2-3 et x, y sont des étiquettes telles que : $\max g \leq x \leq \min m \leq \max m \leq y \leq \min d$.
- et parfaitement équilibré : tous les chemins de la racine à une feuille ont même longueur.

Un nœud ternaire contient donc un fils central dont les valeurs sont comprises entre ses deux étiquettes.

Voici un exemple d'arbre 2-3 :



Théorème IV.1 Un arbre 2-3 à n nœuds est de hauteur au plus $\lfloor \log_2 n \rfloor$.

Démonstration.

Les nœuds ternaires rendent l'arbre compact. Le maximum de hauteur correspond donc au cas où il ne contient que des nœuds binaires sauf sur son dernier niveau, un tel arbre est bien de hauteur $\lfloor \log_2 n \rfloor$. ■

Recherche

Pour rechercher un élément x on adapte l'algorithme des ABRs :

- pour un noeud binaire $N(g, y, d)$
 - * si $x < y$ on cherche dans g ;
 - * si $x = y$ on a trouvé le nœud voulu;
 - * si $x > y$ on cherche dans d .
- pour un noeud ternaire $M(g, y, m, z, d)$
 - * si $x < y$ on cherche dans g ;
 - * si $x = y$ ou $x = z$ on a trouvé le nœud voulu;
 - * si $z > x > y$ on cherche dans m ;
 - * si $z < x$ on cherche dans d .

La complexité de cet algorithme est donc encore en $O(h)$ où h est la hauteur de l'arbre. En tenant compte de la remarque précédente, elle est en $O(\log n)$.

Insertion

Pour insérer un élément x dans un arbre 2-3, on va temporairement enrichir sa structure par des nœuds quaternaires satisfaisant les mêmes requêtes sur les clés que les nœuds binaires et ternaires.

Pour insérer on effectue alors une recherche qui nous amène au dernier nœud avant les feuilles. On transforme alors ce nœud en augmentant son arité et en plaçant x à la bonne position par rapport aux étiquettes qu'il contient.

Exemples : * 1 dans



donne



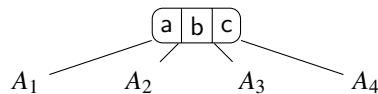
- 8 dans



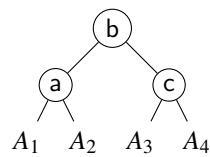
donne



Maintenant on va supprimer les nœuds quaternaires en les faisant remonter, au pire, jusqu'à la racine pour laquelle on peut faire l'opération suivante :



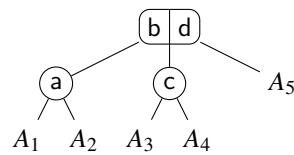
→



Voici tous les cas de transformations selon le type du père et la position du nœud quaternaire : * fils gauche d'un binaire :



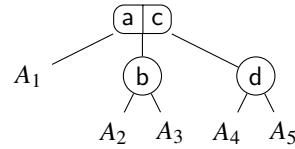
→



- fils droit d'un binaire :



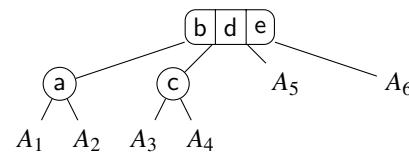
→



- fils gauche d'un ternaire :



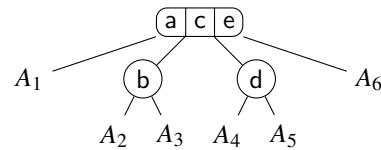
→



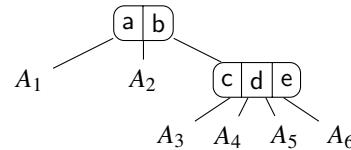
- fils central d'un ternaire :



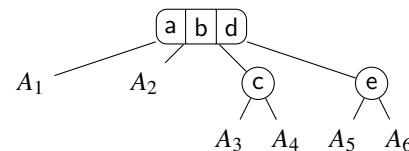
→



- fils droit d'un ternaire :



→



On observe rapidement que toutes ces transformations préservent la position des sous-arbres vis-à-vis des étiquettes des nœuds. Ainsi, au final on aura obtenu un arbre 2-3.

Remarque sur l'implémentation

Les arbres 2-3 sont simples à comprendre mais complexes à programmer :

- beaucoup de cas à traiter, notamment en fonction des parents ;
- des nœuds intermédiaires 2-3-4 qui demandent d'avoir un type de travail et un type final.

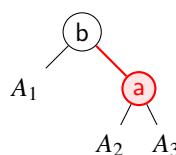
On va voir dans un dernier temps des arbres quasiment équivalents à ceux-ci mais bien plus simples à programmer.

IV.4.iii Arbres rouges et noirs

On va transformer un arbre 2-3 en ABR en transformant les nœuds ternaires ainsi :



→



Afin de se souvenir que le nœud *a* provient de cette séparation, on le marque en rouge. Ici on a également marqué l'arête gauche en rouge mais on l'omettra dans la suite.

Quelles remarques :

- la racine est noire et les feuilles sont noires ;
- le fils gauche d'un nœud est toujours noir ;
- il ne peut exister d'arêtes liant deux nœuds rouges car par construction les nœuds rouges sont toujours entourés de nœuds noirs ;
- les chemins d'une feuille à la racine dans l'arbre 2-3 sont tous de même taille, à chacun des nœuds de l'arbre 2-3 correspond un nœud noir, on en déduit donc que tous les chemins de la racine aux feuilles contiennent le même nombre de nœud noirs.

Réciproquement un arbre ABR dont les nœuds peuvent avoir ces deux couleurs et qui vérifie ces propriétés est appelé un arbre rouge et noir.

En fait, on a une bijection entre les arbres 2-3 et les arbres rouges et noirs ainsi définis.

La proposition suivante permet de voir que les arbres rouges et noirs sont de bons ABRs :

Théorème IV.2 La hauteur d'un arbre rouge et noir est au maximum égale au double de la hauteur de l'arbre 2-3 qui lui est associé.

Ainsi tout arbre rouge et noir de taille n est de hauteur $O(\log n)$.

Démonstration.

La hauteur n'augmente que par séparation des nœuds ternaires. Or, il ne peut il y en avoir plus que la hauteur de l'arbre 2-3.

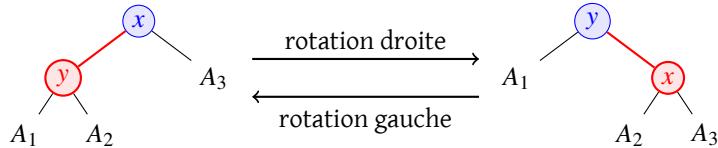
■

Pour l'insertion d'un nœud, on va déstabiliser la coloration en mettant trop de nœuds rouges, puis on va rétablir les contraintes en remontant vers la racine. Cela rappelle l'élimination des nœuds quartenaires des arbres 2-3. Mettre des nœuds rouges à l'avantage de préserver l'équilibrage noir.

Pour rétablir les couleurs, on va alors utiliser une succession de deux opérations élémentaires : les rotations et la bascule.

Rotations

On définit les deux rotations gauche et droite, duales l'une de l'autre, ainsi :



Ici, on a utilisé la couleur bleue pour représenter la couleur du nœud de départ qui peut être soit rouge soit noir mais va être préservée.

Les rotations permettent donc de faire pencher des nœuds rouges à droite ou à gauche.

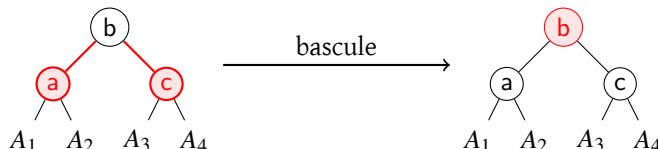
On remarque que la hauteur en nœuds noirs de l'arbre n'a pas changé.

On peut passer d'une des trois représentations convenables des nœuds quaternaires à une autre à l'aide de rotations :



Bascule

Une autre opération importante dans les arbres rouges et noirs est le bascule qui permet de faire remonter les défauts de coloration vers la racine tout en préservant l'équilibre en nœuds noirs :



Si on effectue une bascule directement sur la racine de l'arbre considéré, on finit par la noircir ce qui augmente de un la profondeur noire de chaque feuille mais ne change pas l'équilibrage global.

Insertion

Selon les remarques précédentes, on ne va pas hésiter à perdre la propriété des arbres rouges et noirs temporairement. De toutes les propriétés de ces arbres, c'est sûrement le fait qu'ils soient parfaitement équilibrés en nœuds noirs qui est le plus dur à assurer. On ne va donc pas toucher à cette propriété : on va insérer des nœuds rouges sous une feuille.

En faisant cela on risque d'avoir deux types de problèmes :

- des arêtes rouge-rouge;
- des fils gauches rouges.

On va alors traiter les différents cas d'insertion et voir comment on peut les ajuster pour obtenir des arbres rouges et noirs.

Insertion dans un nœud issu d'un nœud binaire

Un nœud noir issu d'un nœud binaire a nécessairement ses deux fils qui sont vides ou non vides simultanément, car ils proviennent directement de la structure parfaitement équilibré des arbres 2-3.

On a alors les deux cas suivants d'insertion :



Dans les deux cas on se ramène à un nœud ternaire directement.

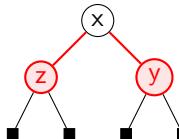
Insertion dans un nœud issu d'un nœud ternaire

On a trois cas d'insertions selon la position de la nouvelle valeur par rapport aux nœuds deux valeurs du nœud ternaire.

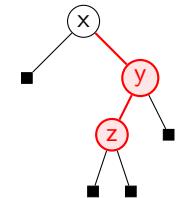
Supposons que l'on insère z dans l'arbre :

- Si $x \leq y \leq z$:
-
- A ternary tree with root x (black circle) having three children: a black child on the left, a red child y in the middle, and a red child z on the right.

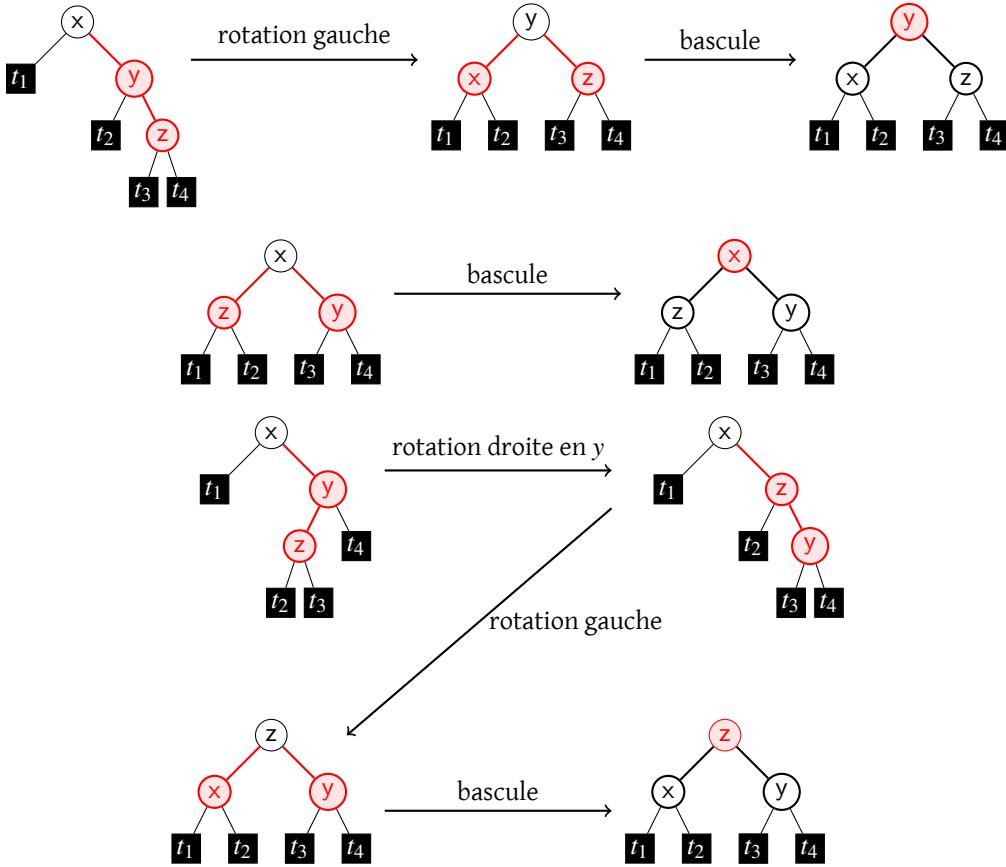
- Si $z \leq x \leq y$:



- Si $x \leq z \leq y$:



Ici, on ne peut pas rétablir tout de suite la structure mais on peut faire remonter le problème jusqu'à la racine, quitte à faire une bascule finale qui va la noircir. On va donc partir d'un des trois problèmes précédents mais supposer qu'il se situe sur un nœud quelconque avec des sous-arbres rouges et noirs t_1, t_2, t_3 et t_4 .



On remarque qu'on effectue essentiellement les mêmes opérations, et quitte à les inhiber quand on ne peut pas les appliquer, on peut se dire que pour insérer z dans l'arbre a , on le rajoute récursivement à gauche ou à droite, on obtient ainsi a' et on effectue alors

$$(\text{bascule}^? \circ \text{rot_gauche}^? \circ \text{rot_droite}^?)(a')$$

où ici, $\text{op}^?$ signifie qu'on effectue l'opération uniquement si elle s'applique :

- pour une rotation droite, il faut que le fils gauche soit rouge et le fils droit noir, pour ne pas tomber dans le cas d'une bascule
- pour une rotation gauche, il faut que le fils droit et le fils droit du fils droit soient rouges
- pour une bascule, que les deux fils soient rouges

Implémentation

On commence par définir le type des arbres rouges et noirs :



On définit ensuite des fonctions qui nous seront utiles vue la remarque précédente pour tester les couleurs à gauche et à droite :



On implémente alors les rotations et la bascule avec le filtrage.



Puis, les versions optionnelles qui ne s'effectuent que si c'est nécessaire :

C

On peut alors réaliser l'insertion récursivement :

C

V Tas

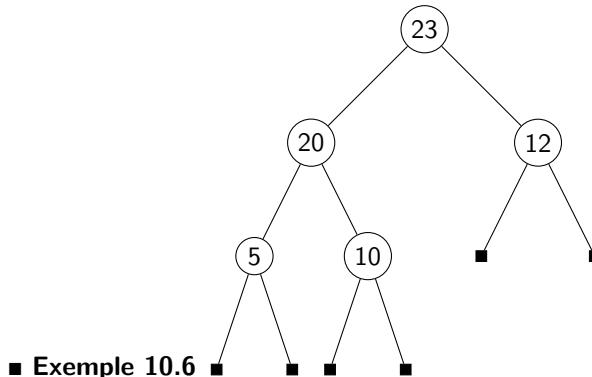
V.1 Présentation

On va considérer une structure de donnée permettant de réaliser efficacement deux opérations :

- récupérer la plus grande des valeurs ajoutées et la supprimer
- ajouter une nouvelle valeur

Avec des arbres, on cherche à réaliser ces opérations en $O(\log n)$ où n est le nombre de valeurs présentes.

Définition V.1 On appelle *tas sur l'ensemble X* un arbre binaire étiqueté par X et complet à gauche tel que pour tout noeud de l'arbre, son étiquette est supérieure ou égale à celle de ses fils.



■ Exemple 10.6 ■

■ **Remarque 10.10** On parle de *heap* en anglais. Ici, nous présentons des tas-max permettant de déduire le maximum, on pourrait définir des tas-min, mais quitte à renverser la relation d'ordre, il s'agit de la même structure. ■

V.2 Opérations

Dans les deux opérations qu'on souhaite implémenter, on va avoir besoin d'un accès rapide à la dernière feuille insérée et la place, à sa droite, de la prochaine feuille qu'on pourrait insérer.

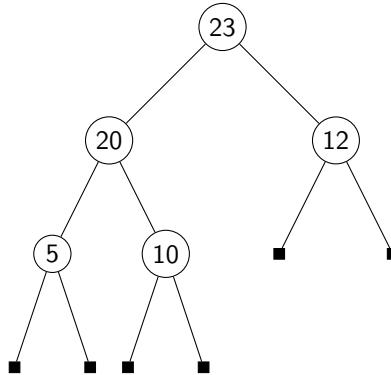
V.2.i Insertion

Pour insérer un élément dans un tas on adopte la stratégie suivante :

- on le place dans la seule place libre permettant de garantir que l'arbre soit complet à gauche;
- on l'insère à la bonne place sur la branche qui le relie à la racine en effectuant des échanges successifs jusqu'à ce que la condition de tas soit vérifiée (étape appelée *remontée du tas*).

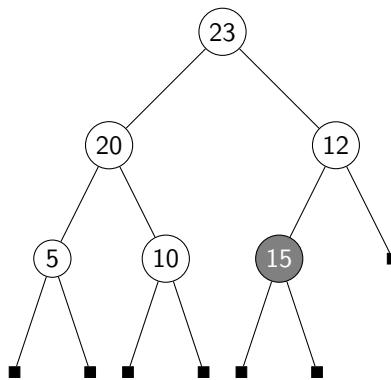
On va illustrer cet algorithme sur un exemple.

On part du tas

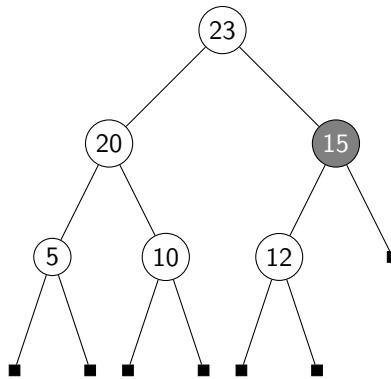


dans lequel on insère la valeur 15.

On commence par ajouter ce nœud dans le premier emplacement libre à droite de 10 :



Comme le fils droit de 12 a une valeur plus grande que celui-ci on échange les étiquettes pour obtenir :



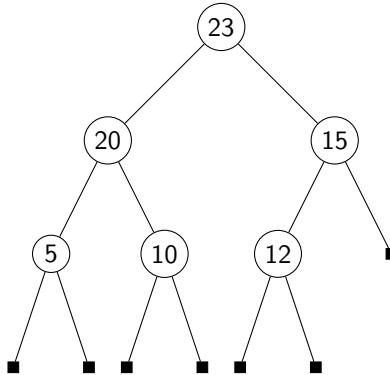
Maintenant le père de 15 est bien plus grand que celui-ci, on s'arrête car on a obtenu un tas.

V.2.ii Extraction du maximum

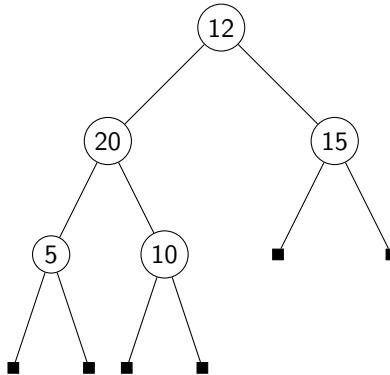
Pour supprimer la racine :

- soit x l'étiquette du nœud situé juste avant le prochain emplacement libre (Il s'agit de la dernière position remplie par une insertion), on place la valeur x à la racine et le nœud où était situé x devient une feuille ;
- on l'insère ensuite à la bonne place à chaque étape avec le plus grand de ses fils, ce qui garantit de prendre un nouveau père respecte bien la condition de tas (étape appelée *descente du tas*).

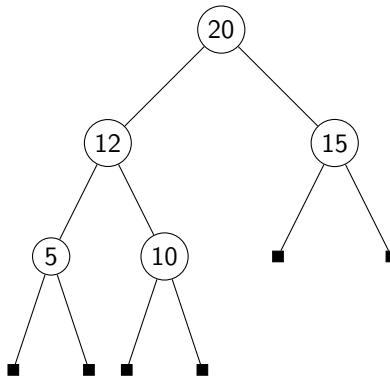
Sur le tas suivant on a $x = 12$



On obtient donc après la première étape l'arbre



On échange alors 12 avec 20 qui est le plus grand des fils :



Maintenant 12 est à la bonne place : on a obtenu un tas.

V.2.iii Complexité

Dans l'hypothèse de pouvoir accéder rapidement (au moins en $O(\log n)$) à la fin du dernier niveau, on peut réaliser ces deux opérations en remontant ou en descendant le long d'une branche, donc en $O(\log n)$.

V.3 Implémentation

On va reposer sur l'implémentation à plat d'un arbre comme décrit précédemment. On va supposer qu'on dispose d'une constante N et que l'on sait que le tas contiendra au maximum N valeurs. Cette implémentation est alors à rapprocher de celle faite pour les files ou les piles dans des tableaux.

```
struct tas {
    int *elements;
    int taille;
    int taille_max;
};
typedef struct tas tas;

tas *tas_cree(int N)
{
    tas *t = malloc(sizeof(tas));
    t->elements = malloc(N * sizeof(int));
    t->taille = 0;
    t->taille_max = N;
}

void tas_libere(tas *t)
{
    free(t->elements);
    free(t);
}

void swap(int *t, int i, int j)
{
    int z = t[i];
    t[i] = t[j];
    t[j] = z;
}

int pere(int i)
{
    return (i-1)/2;
}

int gauche(int i)
{
    return 2*i+1;
}

int droite(int i)
{
    return 2*i+2;
}

void tas_insere(tas *t, int x)
{
    assert(t->taille < t->taille_max);
    int n = t->taille;
    t->taille++;
    t->elements[n] = x;
    while(n > 0 && t->elements[pere(n)] < t->elements[n])
    {
        swap(t->elements, n, pere(n));
        n = pere(n);
    }
}
```

```

}

bool feuille(tas *t, int i)
{
    return gauche(i) >= t->taille;
}

int max_gauche_droite(tas *t, int i)
{
    int vg = t->elements[gauche(i)];
    int vd = t->elements[droite(i)];
    if (vg > vd)
        return vg;
    else
        return vd;
}

int tas_extrait_max(tas *t)
{
    int root = t->elements[0];
    int x = t->elements[t->taille - 1];
    t->elements[0] = x;
    t->taille--;

    int n = 0;
    while(!feuille(t, n) && t->elements[n] < max_gauche_droite(t, n))
    {
        int vg = t->elements[gauche(n)];
        int vd = t->elements[droite(n)];
        if (vg > vd)
        {
            swap(t->elements, n, gauche(n));
            n = gauche(n);
        }
        else
        {
            swap(t->elements, n, droite(n));
            n = droite(n);
        }
    }

    return root;
}

```

V.4 Application au tri

Pour trier, il suffit de créer un tas, d'insérer tous les éléments puis de les extraire un à un. On appelle cela le *tri par tas*. C'est d'ailleurs l'origine historique de la structure de donnée.

Une manière intéressante de faire cet algorithme consiste à le faire en place, c'est-à-dire sans utiliser un autre tableau pour le tri. Pour cela, on va maintenir dans le même tableau le tas en construction en préfixe et les éléments à insérer en suffixe.

Pour reconstruire le tableau trié, il suffit alors d'extraire les maximums et des les placer à la fin du tableau en faisant ici grossir le suffixe trié et réduire le tas préfixe.

```

void heapify(int *t, int sz)
{
    for(int i = 1; i < sz; i++)
    {
        while(i > 0 && t[(i-1)/2] < t[i])
        {
            int p = (i-1)/2;
            swap(t, i, p);
            i = p;
        }
    }
}

void sort(int *t, int sz)
{
    heapify(t, sz);
    for(int i = sz-1; i > 0; i--)
    {
        swap(t, i, 0);
        swap(t, i-1, 0);
        int j = 0;
        while(2*j+1 < i-1 && t[j] < max(t[2*j+1],t[2*j+2]))
        {
            int c = 2*j+2;
            if(t[2*j+1] >= t[2*j+2])
                c = 2*j+1;
            swap(t, j, c);
            j = c;
        }
    }
}

```

V.5 Application aux files de priorité

A la manière des dictionnaires, on peut considérer des couples (*prio, val*) pour les étiquettes en utilisant uniquement la première composante dans le tas. Ainsi, on réalise la structure de données files de priorité introduite dans un chapitre précédent.

VI TP

VI.1 Arbres en OCaml

VI.1.i Premières fonctions

On va considérer le type des arbres binaires '*a arbre*' défini par :

OCaml `type 'a arbre = Nil | Noeud of 'a arbre * 'a * 'a arbre`

Question VI.1 Écrire des fonctions :

OCaml

```
(* Calcule le nombre de noeuds de a *)
let taille (a : 'a arbre) : int
(* Calcule la hauteur de a *)
let hauteur (a : 'a arbre) : int
(* Indique si a est réduit à une feuille *)
let feuille (a : 'a arbre) : bool
```



Démonstration.

OCaml

```
let rec taille a = match a with
| Nil -> 0
| Noeud(g,x,d) -> 1 + taille g + taille d

let rec hauteur a = match a with
| Nil -> -1
| Noeud(g,x,d) ->
    1 + max (hauteur g) (hauteur d)

let feuille a = match a with
| Nil -> false
| Noeud(g, x, d) -> g = Nil && d = Nil
```



Question VI.2 Écrire une fonction `sous_arbres` qui renvoie la liste des sous-arbres non vides d'un arbre.

En déduire des fonctions `noeuds`, `feuilles` et `noeuds_internes` qui renvoient la liste des étiquettes des noeuds correspondants.



Démonstration.

OCaml

```
let rec sous_arbres a = match a with
| Nil -> []
| Noeud(g, x, d) ->
    a :: (sous_arbres g @ sous_arbres d)

let rec noeuds a = match a with
| Nil -> []
| Noeud(g, x, d) ->
    x :: (noeuds g @ noeuds d)

let rec feuilles a = match a with
| Nil -> []
| Noeud(g, x, d) ->
    if feuille a then [x]
    else feuilles g @ feuilles d

let rec noeuds_internes a = match a with
| Nil -> []
| Noeud(Nil, _, Nil) -> []
| Noeud(g, x, d) ->
    x :: (noeuds_internes g @ noeuds_internes d)
```

Pour accéder à un nœud de l'arbre, on descend en partant de la racine et en allant à gauche ou à droite. On peut donc représenter un tel chemin par une liste de déplacements :

```
Ocaml type deplacement = Gauche | Droite
type chemin = deplacement list
```

Question VI.3 Écrire une fonction `chemin_noeud : 'a arbre -> chemin -> 'a option` qui renvoie l'étiquette d'un nœud donné par son chemin. On utilise un type `option` en cas de chemin invalide.

Démonstration.

```
Ocaml type deplacement = Gauche | Droite
type chemin = deplacement list

let rec chemin_noeud a l =
  match a, l with
  | Nil, _ -> None
  | Noeud(g,x,d), Gauche::q -> chemin_noeud g q
  | Noeud(g,x,d), Droite::q -> chemin_noeud d q
  | Noeud(g,x,d), [] -> Some x
```

VI.1.ii Parcours

Sur le même modèle que l'exploration par *backtracking*, on va réaliser un parcours en profondeur d'un arbre en explorant à gauche puis à droite ses sous-arbres de manière récursives. Chaque nœud est donc vu trois fois :

- une première fois quand on le découvre avant d'explorer son sous-arbre gauche
- entre les deux explorations
- enfin quand on a fini d'explorer son sous-arbre droit.

Question VI.4 Écrire des fonctions `affiche_avant`, `affiche_milieu` et `affiche_apres` qui parcourt et affiche les étiquettes d'un `string arbre` selon les trois cas précédents. Note : il s'agit essentiellement du même code à une ligne près.

Démonstration.

```
Ocaml let rec affiche_avant a = match a with
  | Nil -> ()
  | Noeud(g,x,d) ->
    print_string x;
    affiche_avant g;
    affiche_avant d

let rec affiche_milieu a = match a with
  | Nil -> ()
```

```

| Noeud(g,x,d) ->
    affiche_milieu g;
    print_string x;
    affiche_milieu d

let rec affiche_apres a = match a with
| Nil -> ()
| Noeud(g,x,d) ->
    affiche_apres g;
    affiche_apres d;
    print_string x

```

Question VI.5 On peut représenter une expression arithmétique comme $(2 + 3) \times 4$ en tant qu'arbre avec des opérations pour les nœuds internes et des valeurs pour les feuilles. Quitte à utiliser des `string` dans les deux cas, qu'obtient-on sur une telle expression avec les trois opérations précédentes ?

Démonstration.

```

let rec evaluer a = match a with
| Nil -> failwith "Vide"
| Noeud(Nil,x,Nil) -> int_of_string x
| Noeud(g, x, d) ->
    let eg = evaluer g in
    let ed = evaluer d in
    match x with
    | "+" -> eg + ed
    | "*" -> eg * ed
    | _ -> failwith "Opé inconnue"

```

L'appel à `affiche_avant` sur l'expression $(2 + 3) \times 4$ va afficher `*+234`.

L'appel à `affiche_milieu` sur l'expression $(2 + 3) \times 4$ va afficher `2+3*4`.

L'appel à `affiche_apres` sur l'expression $(2 + 3) \times 4$ va afficher `23+4*`.

On retrouve ainsi les notions préfixes, infixes et postfixes d'une expression arithmétique.

On redonne ici l'implémentation d'une pile et d'une file en OCaml :

```

let pile_creer () = ref []
let pile_empile p x = p := x :: !p
let pile_depile p =
  match !p with
  | [] -> failwith "Pile vide"
  | x::q -> p := q; x
let pile_est_vide p = !p = []

```



```

let file_creer () = (pile_creer (), pile_creer ())
let file_bascule (pin, pout) = pout := List.rev !pin; pin := []
let file_enfile (pin, pout) x = pile_empile pin x
let file_defile (pin, pout) =

```

```

if pile_est_vide pout
then file_bascule (pin, pout);
pile_depile pout
let file_est_vide (pin, pout) = pile_est_vide pin && pile_est_vide pout

```

Ces deux structures de données ont la même interface mais deux comportements différents. On va utiliser un unique type pour les représenter afin de permettre à une fonction d'utiliser soit une pile soit une file.

```

type ('a, 'b) taches = {
  creation : unit -> 'a;
  ajouter : 'a -> 'b -> unit;
  retirer : 'a -> 'b;
  est_vide : 'a -> bool
}

let taches_pile = {
  creation = pile_creer;
  ajouter = pile_empile;
  retirer = pile_depile;
  est_vide = pile_est_vide
}

let taches_file = {
  creation = file_creer;
  ajouter = file_enfile;
  retirer = file_defile;
  est_vide = file_est_vide
}

```

■ **Remarque 10.11** Ici, on a une limitation du système de types. On aimeraient paramétriser taches par ('a 'b, 'a) où 'b est un type paramètré, ainsi on aurait soit ('a pile, 'a) où ('a file, 'a) comme pour les deux cas. Cependant, ce n'est pas possible, les types génériques ne sont pas paramétriques. C'est pour cela qu'on a un ('a, 'b) sans avoir de lien apparent entre 'a et 'b.

On pourra alors écrire une fonction prenant un gestionnaire de tâches en paramètre :

```

let f t a =
  let a_traiter = t.creation () in
  for i = 0 to Array.length a - 1 do
    t.ajouter a_traiter a.(i)
  done;
  while not (t.est_vide a_traiter) do
    print_int (t.retirer a_traiter)
  done

```

Ainsi f taches_pile [|1;2;3|] va afficher 321 et f taches_file [|1;2;3|] va afficher 123.

Question VI.6 En déduire une fonction

Ocaml parcours : ('a, 'b arbre) taches -> 'b arbre -> unit

qui effectue un parcours de l'arbre en affichant les étiquettes des nœuds visités dans l'ordre induit par le gestionnaire de tâches passé en paramètre.

Démonstration.

```
let parcours t a =
  let avisiter = t.creation () in
  t.ajouter avisiter a;
  while not (t.est_vide avisiter) do
    match t.retirer avisiter with
    | Nil -> ()
    | Noeud(g,x,d) ->
        print_int x;
        t.ajouter avisiter g;
        t.ajouter visiter d
  done
```

VI.1.iii Arbres binaires stricts

Pour représenter des arbres dont les nœuds ont tous deux fils non vides, on va utiliser le type

Ocaml type ('a, 'b) arbre_bin = Feuille of 'a
| NoeudI of ('a, 'b) arbre_bin * 'b * ('a, 'b) arbre_bin

Ce type permet de représenter naturellement des expressions arithmétiques comme :

Ocaml NoeudI(NoeudI(Feuille 2, '+', Feuille 3), '*', Feuille 5)

Question VI.7 Écrire une fonction evaluate : (int, char) arbre_bin -> int qui évalue une telle expression.

Démonstration.

```
let rec evaluate a = match a with
  | Feuille n -> n
  | NoeudI(g, x, d) ->
      let eg = evaluate g in
      let ed = evaluate d in
      match x with
      | '+' -> eg + ed
      | '*' -> eg * ed
      | _ -> failwith "Ope inconnue"
```

Si on considère le type

Ocaml

```
type ('a, 'b) etiquette = F of 'a | N of 'b
```

On peut passer d'un ('a, 'b) arbre_bin à un ('a, 'b) etiquette arbre et, dans certains cas, d'un ('a, 'b) etiquette arbre à un ('a, 'b) arbre_bin.

Question VI.8 Écrire ainsi deux fonctions de conversion :

Ocaml

```
arbre_bin_vers_arbre : ('a, 'b) arbre_bin -> ('a, 'b) etiquette arbre
arbre_vers_arbre_bin : ('a, 'b) etiquette arbre -> ('a, 'b) arbre_bin option
```

Démonstration.

Ocaml

```
let rec arbre_bin_vers_arbre a =
  match a with
  | Feuille n -> Noeud(Nil, F n, Nil)
  | NoeudI(g, x, d) -> Noeud(arbre_bin_vers_arbre g,
                                N x,
                                arbre_bin_vers_arbre d)

let rec arbre_vers_arbre_bin a =
  match a with
  | Noeud(Nil, F f, Nil) -> Feuille f
  | Noeud(g, N x, d) -> begin
    match arbre_vers_arbre_bin g, arbre_vers_arbre_bin d with
    | Some g', Some d' -> NoeudI(g', x, d')
    | _ -> None
  end
  | _ -> None
```

VI.1.iv Dessin d'arbres

On va réaliser dans cette partie une fonction de dessin d'arbres avec `graphics`. L'idée est de placer la racine et de dessiner les sous-arbres gauche et droit en dessous. Pour cela, il va falloir connaître la taille de ces sous-arbres en pixels.

On pourra consulter la documentation de `graphics` ici : [Graphics](#)

Question VI.9 Écrire une fonction `pixels : int arbre -> int * int` qui renvoie un couple (`largeur, racine`) où `largeur` est la largeur en pixels d'un arbre dans son affichage et `racine` l'abscisse de la racine dans cet affichage.

Démonstration.

Ocaml

```
let rnoeud = 5 (* Le rayon du noeud *)
let marge = 2 (* la marge autour du noeud *)

let rec pixels a = match a with
  | Nil -> 0, 0
  | Noeud(g, _, d) ->
```

```

let lg, _ = pixels g in
let ld, _ = pixels d in
lg + ld + 2*(marge + rnoeud), lg + marge + rnoeud

```

Question VI.10 En déduire une fonction dessine : int arbre → int → int → unit tel que dessine a x y dessine l'arbre a en plaçant la racine en (x, y).

Démonstration.

```

let rec dessine a x y =
  match a with
  | Nil -> ()
  | Noeud(g, e, d) ->
    (* On précalcule les tailles des sous-arbres *)
    let lg, rg = pixels g in
    let _, rd = pixels d in
    (* position de la racine gauche relativement à (x,y) *)
    let v_rg = x - lg - marge - rnoeud + rg in
    (* position de la racine droite relative (x, y) *)
    let v_rd = x + marge + rnoeud + rd in
    let dec_y = y - 2 * rnoeud - marge in

    moveto x y; lineto v_rg dec_y;
    moveto x y; lineto v_rd dec_y;
    fill_circle x y rnoeud;

    moveto (x-tw/2) (y-th/2);
    set_color white;
    draw_string e;
    set_color black;

    dessine g v_rg dec_y;
    dessine d v_rd dec_y

```

Ocaml

VI.1.v Génération aléatoire d'arbres

Dans cette partie, ce qui nous intéresse est de générer aléatoirement des arbres pour en observer la structure. Les étiquettes ne sont pas pertinentes, on pourra donc au choix, soit considérer des unit arbre, soit redéfinir un type d'arbre non étiquetés.

On va réaliser ici deux modèles de génération aléatoire d'arbres. Le premier modèle consiste, pour générer un arbre de n noeuds, à choisir aléatoirement k dans $\llbracket 0, n \rrbracket$, à générer aléatoirement un arbre g à k noeuds et un arbre d à $n - k$ noeuds puis à renvoyer l'arbre $\text{Noeud}(g, (), d)$.

Question VI.11 Implémenter ce modèle avec une fonction genere_arbre_1 : int → unit arbre.

Tester votre fonction, notamment avec l'affichage de la partie précédente.

Démonstration.

```
OCaml let rec genere_arbre_1 n =
  if n = 0
  then Nil
  else if n = 1 then Noeud(Nil, (), Nil)
  else let k = Random.int (n-1) in
    Noeud(genere_arbre_1 k, (), genere_arbre_1 (n-1-k))
```

Pour le deuxième modèle, on va choisir un sous-arbre vide uniformément et le remplacer par un nœud.

Question VI.12 Écrire une fonction `chemins_vides` : `'a arbre -> chemin list` qui étant donné un arbre renvoie la liste des chemins permettant d'aboutir à un sous-arbre vide.

Démonstration.

```
OCaml let chemins_vide a =
  let rec aux a l =
    match a with
    | Nil -> [ List.rev l ]
    | Noeud(g, _, d) ->
        aux g (Gauche::l)
        @ aux d (Droite::l)
  in aux a []
```

Question VI.13 Écrire une fonction `rempli` : `'a arbre -> chemin -> 'a arbre` qui remplace un sous-arbre vide donnée par son chemin par un nœud.

Démonstration.

```
OCaml let rec rempli a c =
  match a, c with
  | Nil, [] -> Noeud(Nil, (), Nil)
  | Noeud(g,x,d), Gauche::q ->
      Noeud(rempli g q, x, d)
  | Noeud(g,x,d), Droite::q ->
      Noeud(g, x, rempli d q)
  | _ -> failwith "Error"
```

Question VI.14 En déduire une fonction `genere_arbres_2` : `int -> unit arbre` qui prend un entier n et construit un arbre à n nœuds en partant de l'arbre vide et en réalisant n remplacement d'un arbre vide par un nœuds, le choix de l'arbre vide se faisant uniformément.

Note : on pourra utiliser `List.nth` et `Random.int`. Voir par exemple `Random`

Démonstration.

```
let rec genere_arbre_2 n =
  if n = 0 then Nil
  else
    let a = genere_arbre_2 (n-1) in
    let l = chemins_vide a in
    let k = Random.int (List.length l) in
    let ch = List.nth l k in
    remplit a ch
```



VI.2 Arbres non binaires, tries

VI.2.i Arbres non binaires

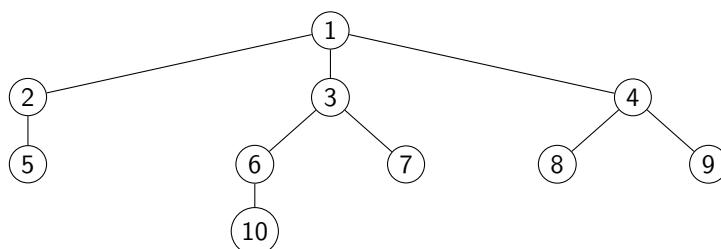
On va considérer ici les arbres d'arité quelconque avec des listes d'enfants pour les noeuds.

```
type 'a arbre = { etiquette : 'a; enfants : 'a arbre list }
```

Afin d'effectuer des tests, on pourra considérer l'arbre suivant :

```
let t = { etiquette = 1;
  enfants = [
    { etiquette = 2; enfants = [
      { etiquette = 5; enfants = [] }
    ] };
    { etiquette = 3; enfants = [
      { etiquette = 6; enfants = [
        { etiquette = 10; enfants = [] }
      ] };
      { etiquette = 7; enfants = [] }
    ] };
    { etiquette = 4; enfants = [
      { etiquette = 8; enfants = [] };
      { etiquette = 9; enfants = [] }
    ] }
  ] }
```

qui correspond à la représentation :



Comme présenté dans ce chapitre, on peut réaliser des fonctions mutuellement récursives pour travailler avec ces arbres et sur la forêt induite par les enfants d'un noeud :

```
Ocaml let rec taille_arbre a =
  1 + taille_foret a.enfants
and taille_foret l = match l with
| [] -> 0
| t::q -> taille_arbre t + taille_foret q
```

Question VI.15 Écrire une fonction `hauteur_arbre : 'a arbre -> int` qui calcule la hauteur d'un arbre sur le modèle de la fonction précédente. On fera attention au fait qu'il n'est pas possible de représenter l'arbre vide mais la forêt vide jouera le même rôle vis-à-vis de `-1`.

Démonstration.

```
Ocaml let rec hauteur_arbre a =
  1 + hauteur_foret a.enfants
and hauteur_foret l = match l with
| [] -> -1
| t::q -> max (hauteur_arbre t) (hauteur_foret q)
```

Question VI.16 A l'aide d'un parcours récursif, écrire une fonction `affiche_etiquettes : int tree -> unit` qui affiche toutes les étiquettes d'un `int tree`.

Démonstration.

```
Ocaml let rec affiche_etiquettes a =
  Printf.printf "%d\n" a.etiquette;
  affiche_foret a.enfants
and affiche_foret l = match l with
| [] -> ()
| t::q -> affiche_etiquettes t; affiche_foret q
```

VI.2.ii Représentation par des arbres binaires

On va maintenant revenir ici sur la représentation d'un '`a arbre`' en tant qu'arbre binaire. Pour cela, on va avoir besoin d'étiquettes factices pour les nouveaux nœuds qui ne servent qu'à supporter la structure des nœuds d'arité > 2 . On va ainsi les représenter par des `a option arbre_bin` où l'étiquette sera celle des nœuds factices.

```
Ocaml type 'a arbre_bin = Noeud of 'a arbre_bin * 'a * 'a arbre_bin | Nil
```

Question VI.17 On va commencer par écrire une fonction `peigne : 'a option arbre_bin list -> 'a option arbre_bin` qui va transformer une forêt d'arbres binaires en sa représentation en peigne à l'aide de nœuds d'étiquettes `None`.

Note Dans le cas où la liste est vide, on renverra l'arbre vide et si c'est un singleton, on

renverra directement l'arbre qu'il contient.

Démonstration.

```
OCaml let rec peigne l =
  match l with
  | [] -> Nil
  | [x] -> x
  | a::q -> Noeud(a, None, peigne q)
```

Question VI.18 Écrire une fonction convert : 'a arbre -> 'a option arbre_bin qui va réaliser la conversion vue en classe.

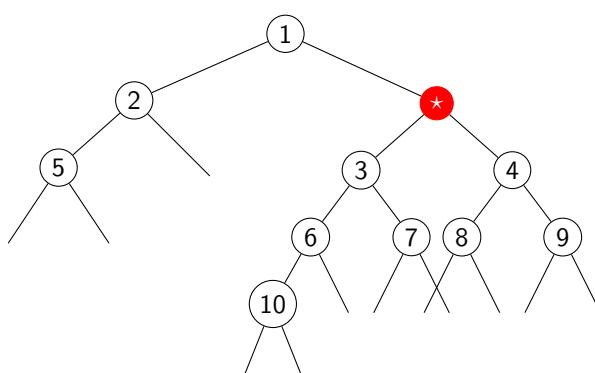
Démonstration.

```
OCaml let rec convert a =
  match a.enfants with
  | [] -> Noeud (Nil, Some a.etiquette, Nil)
  | a1::q -> Noeud(convert a1, Some a.etiquette,
                     peigne (List.map convert q))
```

Afin de tester votre fonction, on vous donne une fonction d'affichage textuel d'un tel arbre :

```
OCaml let rec print_arbre_bin pref t =
  match t with
  | Nil -> Printf.printf "%snil\n%"! pref
  | Noeud(g,x,d) -> Printf.printf "%s%s\n%"! pref
    (match x with None -> "*" | Some n -> string_of_int n);
    print_arbre_bin (String.make (String.length pref) ' '
                      ^ "|" ^ String.make 5 '-' ) g;
    print_arbre_bin (String.make (String.length pref) ' '
                      ^ "|" ^ String.make 5 '-' ) d
```

L'affichage doit correspondre à la représentation suivante :



VI.3 Tries

On va reprendre les tries vus plus haut pour représenter un ensemble de mots. Pour cela, on réutilise le type :

```
OCaml | type trie = {
          mot : bool;
          enfants : (char * trie) list
        }
```

VI.3.i Premières manipulations

Question VI.19 Écrire une fonction `taille : trie -> int` qui renvoie le nombre de nœuds d'un trie et une fonction `nb_mots : trie -> int` qui renvoie le nombre de mots d'un trie. On s'inspirera directement des fonctions de la première partie du TP.

Démonstration.

```
OCaml | let rec taille a =
          1 + taille_foret a.enfants
        and taille_foret l = match l with
          | [] -> 0
          | (_, t) :: q -> taille t + taille_foret q

        let rec nb_mots a =
          (if a.mot then 1 else 0) + nb_mots_foret a.enfants
        and nb_mots_foret l = match l with
          | [] -> 0
          | (_, t) :: q -> nb_mots t + nb_mots_foret q
```

Question VI.20 Écrire une fonction `trie_apres_char` tel que l'appel à `trie_apres_char l c` va renvoyer `Some t'` si il y a un couple (c, t') dans la liste `l` et `None` sinon. On pourra s'en servir avec `trie_apres_char t.enfants c` pour trouver le trie potentiel dont l'arête est étiquetée par `c`.

Démonstration.

```
OCaml | let rec trie_apres_char (l:(char * trie) list) (c:char) =
          match l with
          | [] -> None
          | (c',t)::q -> if c = c' then Some t else trie_apres_char q c
```

Question VI.21 En déduire une fonction `contient : trie -> string -> bool` qui teste si un mot est présent dans le trie.

Démonstration.

```
OCaml
let contient (t:trie) (s:string) : bool =
  let rec aux t i =
    if i = String.length s
    then t.mot
    else match trie_apres_char t.enfants s.[i] with
      | None -> false
      | Some t' -> aux t' (i+1)
  in aux t 0
```

Question VI.22 Écrire une fonction enumere : trie -> string list qui renvoie tous les mots contenus dans un trie par un parcours récursif.

Indication : on pourra utiliser :

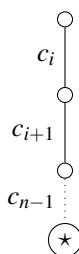
- List.concat pour passer d'une liste de listes à la concaténée de ses éléments. Ainsi List.concat [[1;2]; [3]; [4;5]] va renvoyer [1;2;3;4;5].
- String.make 1 c transforme le caractère c en chaîne de caractère
- On rappelle que ^ permet de concaténer deux chaînes.

Démonstration.

```
OCaml
let enumere (t:trie) =
  let rec aux t acc =
    let l = List.concat (List.map (fun (c,t') ->
        aux t' (acc ^ String.make 1 c)) t.enfants) in
    if t.mot then acc :: l else l
  in aux t ""
```

VI.3.ii Ajout d'un mot à un trie

Pour effectuer l'ajout d'un mot à un trie, on va commencer par écrire une fonction qui étant donné une chaîne de caractère $s = c_0 \dots c_{n-1}$ et un entier $i \leq n$ va renvoyer le trie t correspondant à la branche



Question VI.23 écrire la fonction suffixe : int -> string -> trie

Démonstration.

```
OCaml
let rec suffixe (i:int) (s:string) : trie =
  let n = String.length s in
  if i = n
```

```

  then { mot = true; enfants = [] }
else { mot = false; enfants = [ (s.[i], suffixe (i+1) s) ] }

```

Question VI.24 Écrire une fonction avec_trie_pour_char : (char * trie) list -> char -> trie -> (char * trie) list telle que avec_trie_pour_char l c t va renvoyer la liste obtenue en remplaçant (c, t') par (c, t) dans l si un tel couple est présent, ou en ajoutant le couple (c, t) sinon à la fin.

Démonstration.

```

OCaml | let rec avec_trie_pour_char l c t =
        match l with
        | [] -> [ (c, t) ]
        | (c', t') :: q -> if c = c' then (c, t) :: q else (c', t') :: avec_trie_pour_char q

```

Question VI.25 Écrire une fonction ajoute : trie -> string -> trie qui renvoie le trie obtenu en ajoutant un nouveau mot.

Démonstration.

```

OCaml | let ajoute t s =
        let n = String.length s in
        let rec aux t i =
            if i = n
            then { t with mot = true }
            else let t' = match trie_apres_char t.enfants s.[i] with
                  | None -> suffixe (i+1) s
                  | Some t' -> aux t' (i+1) in
                  { t with enfants = avec_trie_pour_char t.enfants s.[i] t' }
        in aux t 0

```

On va utiliser la fonction précédente pour retrouver le trie donné en début de partie.

```

OCaml | let rec trie_of_list l = match l with
        | [] -> { mot = false; enfants = [] }
        | s::q -> ajoute (trie_of_list q) s

```

On va maintenant récupérer une grande liste de mots dans le fichier wordlist récupérable à l'adresse https://raw.githubusercontent.com/dwyl/english-words/master/words_alpha.txt à raison d'un mot par ligne :

```

OCaml | let wordlist = let rec lit_fichier f =
           try
             let s = input_line f in
             s :: lit_fichier f
           with End_of_file -> []

```

```
with End_of_file -> []
in
lit_fichier (open_in "wordlist")
```

Question VI.26 En déduire un trie permettant de représenter cette wordlist et comparez :

- Le nombre de caractère total
 - Le nombre de noeuds du trie
- Que peut-on en déduire ?

Démonstration.

OCaml

```
let taille_totale = List.fold_left (+) 0 (List.map String.length wordlist) in
taille_totale, taille (trie_of_list wordlist)
```

On voit qu'on 595775 caractères au total pour 229239 noeuds. Attention cependant à ne pas sous-estimer la taille d'un noeud qui dépasse très largement celle d'un caractère. Cependant, le trie apporte la possibilité de chercher un mot m en $O(|m|)$ et donc indépendamment du nombre de mots.



VI.4 Arbres binaires en C, ABR, arbres rouges et noirs

VI.4.i Arbres binaires en C

En C, un nœud d'un arbre binaire va avoir exactement la même structure qu'un maillon d'une liste doublement chaînée : une valeur et deux pointeurs vers d'autres nœuds, ici ce ne sont pas suivant et précédent mais gauche et droite :

```
struct noeud {
    struct noeud *gauche;
    struct noeud *droite;
    int valeur;
};

typedef struct noeud *arbre;
```

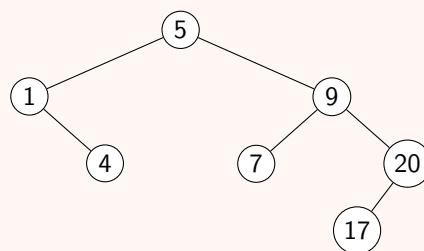
Comme pour les listes, l'arbre vide sera représenté par le pointeur NULL.

Question VI.27 Écrire une fonction arbre_cree_feuille(int x) qui alloue et crée une feuille de valeur x.

Démonstration.

```
arbre cree_feuille(int x)
{
    arbre n = malloc(sizeof(struct noeud));
    n->valeur = x;
    n->gauche = n->droite = NULL;
    return n;
}
```

Question VI.28 En déduire, une représentation en C de l'arbre :



Pour cela, on vous suggère de créer autant de feuilles que de nœuds et d'effectuer les liens ensuite avec les pointeurs.

Cet arbre nous servira d'exemple pour vérifier les fonctions suivantes.

Démonstration.

```
arbre n5 = cree_feuille(5);
arbre n1 = cree_feuille(1);
arbre n4 = cree_feuille(4);
```

```

arbre n9 = cree_feuille(9);
arbre n7 = cree_feuille(7);
arbre n20 = cree_feuille(20);
arbre n17 = cree_feuille(17);
n5->gauche = n1;
n1->droite = n4;
n5->droite = n9;
n9->gauche = n7;
n9->droite = n20;
n20->gauche = n17;

arbre a = n5;

```

Question VI.29 Écrire une fonction `int taille(arbre a)` qui renvoie le nombre de noeuds d'un arbre binaire.

Démonstration.

```

int taille(arbre a)
{
    if (a == NULL)
        return 0;
    return 1 + taille(a->gauche) + taille(a->droite);
}

```

Question VI.30 Écrire une fonction `int hauteur(arbre a)` qui renvoie la hauteur d'un arbre binaire.

Démonstration.

```

int max(int a, int b)
{
    if (a > b) return a;
    else return b;
}

int hauteur(arbre a)
{
    if (a == NULL)
        return -1;
    return 1 + max(hauteur(a->gauche), hauteur(a->droite));
}

```

Question VI.31 Écrire des fonctions `void affiche_prefixe(arbre a)`, `affiche_infixe` et `affiche_postfixe` qui affiche les valeurs des noeuds de l'arbre `a` selon le traitement cor-

respondant dans un parcours.

Écrire une fonction `void affiche(arbre a)` qui, à l'aide des trois traitements, permet d'afficher un arbre avec des triplets comme nœuds. Par exemple, pour l'arbre précédent, on pourra afficher :

```
((nil,1,(nil,4,nil)),5,((nil,7,nil),9,((nil,17,nil),20,nil)))
```

Démonstration.

```
void affiche_prefixe(arbre a)
{
    if (a != NULL)
    {
        printf("%d", a->valeur);
        affiche_prefixe(a->gauche);
        affiche_prefixe(a->droite);
    }
}

void affiche_infixe(arbre a)
{
    if (a != NULL)
    {
        affiche_infixe(a->gauche);
        printf("%d", a->valeur);
        affiche_infixe(a->droite);
    }
}

void affiche_postfixe(arbre a)
{
    if (a != NULL)
    {
        affiche_postfixe(a->gauche);
        affiche_postfixe(a->droite);
        printf("%d", a->valeur);
    }
}

void affiche(arbre a)
{
    if (a != NULL)
    {
        printf("(");
        affiche(a->gauche);
        printf(",%d,", a->valeur);
        affiche(a->droite);
        printf(")");
    } else printf("nil");
}
```

Question VI.32 (Optionnel en première lecture) À l'aide d'une file dans un tableau de longueur $|a|$, réalisez une fonction void `affiche_largeur(arbre a)` qui affiche les valeurs des nœuds de l'arbre `a` lors d'un parcours en largeur.

Démonstration.

```
void affiche_largeur(arbre a)
{
    arbre *file = malloc(sizeof(arbre) * taille(a));
    file[0] = a;
    int debut = 0;
    int fin = 1;
    while(fin > debut)
    {
        arbre r = file[debut];
        debut++;
        printf("%d", r->valeur);
        if(r->gauche != NULL)
        {
            file[fin] = r->gauche;
            fin++;
        }
        if(r->droite != NULL)
        {
            file[fin] = r->droite;
            fin++;
        }
    }
}
```

Question VI.33 Écrire une fonction void `libere(arbre a)` qui libère tous les nœuds de l'arbre `a`. On fera attention à l'ordre des libérations avec `free` par rapport aux déréférencements de structure (les `->`).

Démonstration.

```
void libere(arbre a)
{
    if (a != NULL)
    {
        libere(a->gauche);
        libere(a->droite);
        free(a);
    }
}
```

VI.4.ii Arbres binaires de recherche

On conserve le type précédent, mais on considère maintenant qu'on a des arbres binaires de recherche.

Question VI.34 Écrire une fonction `arbre recherche(arbre a, int x)` qui renvoie le premier nœud rencontré d'étiquette x dans un ABR ou `NULL` s'il n'est pas présent.

Démonstration.

```
arbre recherche(arbre a, int x)
{
    if (a == NULL)
        return NULL;
    if (a->valeur == x)
        return a;
    if (a->valeur < x)
        return recherche(a->droite, x);
    return recherche(a->gauche, x);
}
```

Question VI.35 Écrire une fonction `arbre insere(arbre a, int x)` qui renvoie la racine de l'ABR obtenu en insérant une nouvelle feuille d'étiquette x .

Démonstration.

```
arbre insere(arbre a, int x)
{
    if (a == NULL)
        return cree_feuille(x);

    if (a->valeur >= x)
    {
        a->gauche = insere(a->gauche, x);
    }
    else
    {
        a->droite = insere(a->droite, x);
    }

    return a;
}
```

Question VI.36 Écrire une fonction `arbre minimum(arbre a)` qui renvoie le nœud de valeur minimale de a en allant le plus à gauche dans l'arbre a . On supposera a non vide.

Démonstration.

```

C arbre minimum(arbre a)
{
    if (a->gauche == NULL)
        return a;
    return minimum(a->gauche);
}

```

Question VI.37 (Optionnel en première lecture) Écrire une fonction `arbre supprime(arbre a, int x)` qui renvoie la racine de l'ABR obtenu en supprimant le premier nœud rencontré d'étiquette `x`.

Démonstration.

```

C arbre supprime(arbre a, int x)
{
    if (a == NULL)
        return a;
    if (a->valeur == x)
    {
        if (a->gauche == NULL && a->droite == NULL)
        {
            free(a);
            return NULL;
        }
        arbre min_a = minimum(a);
        if (min_a == a)
        {
            arbre a_p = a->droite;
            free(a);
            return a_p;
        }
        int min_a_valeur = min_a->valeur;
        supprime(a, min_a_valeur);
        a->valeur = min_a_valeur;
    } else {
        if (a->valeur > x)
            a->gauche = supprime(a->gauche, x);
        else
            a->droite = supprime(a->droite, x);
    }
    return a;
}

```

VI.4.iii Arbres rouges et noirs

On va rajouter un champ au type précédent pour indiquer la couleur d'un nœud :

```

struct noeud {
    struct noeud *gauche;
    struct noeud *droite;
    bool rouge;
    int valeur;
};
typedef struct noeud *arbre;

```

On vous conseille ici de copier le fichier précédent et de faire un nouveau programme.

Question VI.38 Reprendre `cree_feuille` pour que les feuilles soient rouges par défaut.

Démonstration.

```

arbre cree_noeud(int x)
{
    arbre n = malloc(sizeof(struct noeud));
    n->valeur = x;
    n->rouge = true;
    n->gauche = n->droite = NULL;
    return n;
}

```

Question VI.39 Écrire des fonctions `arbre rotation_droite(arbre a)`, `arbre rotation_gauche(arbre a)` et `arbre bascule(arbre a)` qui effectuent une des trois opérations en renvoyant la nouvelle racine.

Démonstration.

```

arbre rotation_droite(arbre x)
{
    arbre y = x->gauche;
    x->gauche = y->droite;
    y->droite = x;
    y->rouge = x->rouge;
    x->rouge = true;
    return y;
}

arbre rotation_gauche(arbre y)
{
    arbre x = y->droite;
    y->droite = x->gauche;
    x->gauche = y;
    x->rouge = y->rouge;
    y->rouge = true;
    return x;
}

arbre bascule(arbre a)

```

```

{
    a->rouge = true;
    a->gauche->rouge = false;
    a->droite->rouge = false;
    return a;
}

```



Question VI.40 Écrire des fonctions utilitaires `bool rouge(arbre a)` et `bool noir(arbre a)` qui teste la couleur de la racine. On fera attention au fait que l'arbre vide est noir.



Démonstration.

```

bool rouge(arbre a)
{
    if (a == NULL) return false;
    return a->rouge;
}

bool noir(arbre a)
{
    return !rouge(a);
}

```



Question VI.41 Écrire des fonctions `arbre essai_rotgauche(arbre a)`, `essai_rotdroite` et `essai_bascule` qui n'effectuent ces opérations que dans les cas nécessaires pour l'insertion.



Démonstration.

```

arbre essai_bascule(arbre a)
{
    if(rouge(a->gauche) && rouge(a->droite))
        return bascule(a);
    return a;
}

arbre essai_rotgauche(arbre a)
{
    if(rouge(a->droite) && rouge(a->droite->droite))
        return rotation_gauche(a);
    return a;
}

arbre essai_rotdroite(arbre a)
{
    if(rouge(a->gauche) && noir(a->droite))
        return rotation_droite(a);
    return a;
}

```



}



Question VI.42 En déduire une fonction `arbre insere(arbre a, int x)` qui effectue l'insertion dans un arbre rouge et noir. On pourra définir une fonction auxiliaire `arbre insere_aux(arbre a, int x)` qui effectue l'insertion et qu'on appellera dans `insere` avant de faire le noircissement final de la racine.

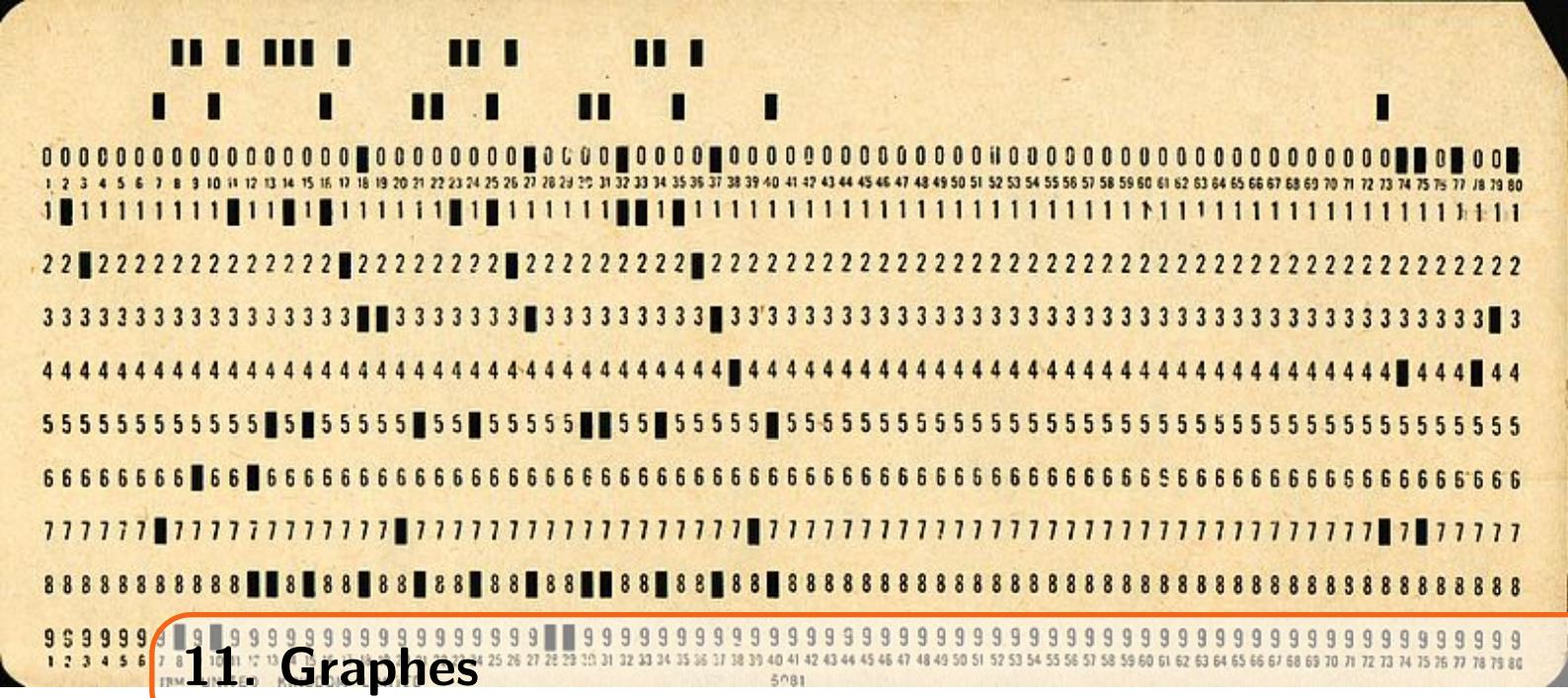


Démonstration.

```
arbre insere_aux(arbre a, int x)
{
    if (a == NULL)
        return cree_noeud(x);
    if (a->valeur > x)
        a->gauche = insere_aux(a->gauche, x);
    else
        a->droite = insere_aux(a->droite, x);
    return essai_bascule(essai_rotgauche(essai_rotdroite(a)));
}

arbre insere(arbre a, int x)
{
    arbre a2 = insere_aux(a, x);
    a2->rouge = false;
    return a2;
}
```





11. Graphes

■ **Note 11.1** Une grande partie est issue verbatim de mon ancien poly. À affiner.

Graphes orientés

1.1 Définition

Définition I.1 Un graphe orienté est un couple $G = (S, A)$ où S est un ensemble fini et A est une relation **irréflexive** sur S , c'est-à-dire $S \subset \{(x, y) \mid x, y \in S, x \neq y\}$.

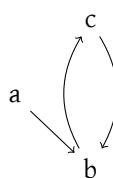
Les éléments de S sont appelés les **sommets** du graphe G et les éléments de A les **arêtes**. Si $(x, y) \in A$, on dit que x est la source de l'arête et y est son but. Quand le contexte n'est pas ambigu, on notera $x \rightarrow y$.

- **Exemple 11.1**
 - $G = (\{a, b, c\}, \{(a, b), (b, c), (c, b)\})$
 - $D_n = ([1, n], \{(a, b) \mid a \text{ divise } b\})$

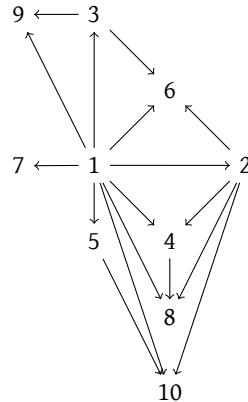
■ **Remarque 11.1** Ici, on a exclu les graphes infinis ainsi que les graphes avec des boucles ou des arêtes parallèles. On est dans le contexte des **graphes finis simples**.

On représente graphiquement un graphe sous la forme d'un diagramme sagittal où les sommets sont des points et où une arête (a, b) est une flèche allant du point a au point b .

- ### ■ Exemple 11.2



- D_{10} est représenté par



I.2 Voisins et degrés

Définition I.2 Soit $G = (S, A)$ un graphe orienté et $x \in S$.

On appelle **voisins sortants**, ou **successeurs**, de x les éléments de

$$v_+(x) = \{ y \in S \mid (x, y) \in A \}$$

et on appelle **degré sortant** de x le cardinal de cet ensemble $d_+(x) = |v_+(x)|$.

De même, on parle de **voisins entrants**, ou **prédecesseurs** pour les éléments de

$$v_-(x) = \{ z \in S \mid (z, x) \in A \}$$

et on parle de **degré entrant** pour $d_-(x) = |v_-(x)|$.

Théorème I.1 Soit $G = (S, A)$ un graphe, on a $\sum_{x \in S} d_+(x) = \sum_{x \in S} d_-(x) = |A|$

Démonstration.

On peut partitionner A en regroupant les arêtes de même source, on a ainsi :

$$A = \bigcup_{x \in S} \{ (x, y) \mid y \in S, (x, y) \in A \} = \bigcup_{x \in S} \{ (x, y) \mid y \in v_+(x) \}$$

Et en prenant la cardinal de cette égalité, on obtient directement $|A| = \sum_{x \in S} d_+(x)$. L'autre égalité est symétrique en considérant les arêtes de même but.

I.3 Chemin

Définition I.3 Soit $G = (S, A)$ un graphe orienté et $x, y \in S$.

Une suite finie $\varphi = (s_0, s_1, \dots, s_p)$ où $p \in \mathbb{N}$, $s_0 = x$, $s_p = y$ et

$$\forall i \in \llbracket 1, p \rrbracket, (s_{i-1}, s_i) \in A$$

est appelée un **chemin**, de longueur p , de x vers y . On a donc

$$s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_p$$

On notera $\varphi : x \rightsquigarrow y$ pour signifier que φ est un tel chemin et $x \rightsquigarrow y$ pour signifier qu'il existe un chemin de x vers y . On dit alors que y est accessible depuis x .

Théorème I.2 \rightsquigarrow est la plus petite relation sur S contenant \rightarrow et qui soit réflexive et transitive.

Démonstration.

On considère $\rightarrow \subset R \subset S^2$ telle que R soit réflexive et transitive.

Soient $x, y \in S$ et $\varphi = (s_0, \dots, s_p) : x \rightsquigarrow y$.

Comme $s_{i-1} \rightarrow s_i$ on a $s_{i-1}Rs_i$ et, par transitivité, $x = s_0Rs_p = y$. Donc $x \rightsquigarrow y \Rightarrow xRy$. Ainsi $\rightsquigarrow \subset R$. ■

On dit que \rightsquigarrow est la clôture réflexive et transitive de \rightarrow .

■ **Remarque 11.2** Vocabulaire additionnel :

- Si tous les sommets sont distincts, sauf éventuellement le premier et le dernier, on dit que le chemin est **élémentaire**.
- Si toutes les arêtes sont distinctes, on dit que le chemin est **simple**.
- Si $x = y$, on dit que le chemin est **fermé**.
- Un chemin simple dont tous les sommets sont distincts est appelé une **chaîne**.
- Un chemin élémentaire fermé simple de longueur au moins 1 est appelé un **cycle**. *Comme le chemin vide issu de x est fermé, il est nécessaire de considérer des chemins non vides.*
- Un graphe contenant au moins un cycle est dit **cyclique**. Dans le cas contraire, on dit qu'il est **acyclique**.

■ **Remarque 11.3** Si $\varphi : x \rightsquigarrow y$ et $\psi : y \rightsquigarrow z$ on note $\varphi\psi : x \rightsquigarrow z$ la concaténée des deux chemins. ■

Définition I.4 On note \leftrightarrow la plus grande relation d'équivalence incluse dans \rightsquigarrow .

Plus précisément, on a

$$x \leftrightarrow y \iff (x \rightsquigarrow y \wedge y \rightsquigarrow x)$$

Les classes d'équivalences pour \leftrightarrow sont appelées les composantes **fortement connexes** du graphe. S'il n'y a qu'une classe, on dit que le graphe est **fortement connexe**.

I.4 Sous-graphe

Définition I.5 Soit $G = (S, A)$ et $X \subset S$. On appelle **sous-graphe** induit par X le graphe (X, A_X) où

$$A_X = \{ (a, b) \in A \mid a \in X \wedge b \in X \}$$

Un graphe G' est un **sous-graphe** de G quand c'est le sous-graphe de G induit par les sommets de G' (ils sont alors nécessairement des sommets de G).

■ **Remarque 11.4** Les composantes fortement connexes sont les sous-graphes fortement connexes maximaux pour l'inclusion.

1.5 Implémentation

1.5.i Matrice d'adjacence

Définition 1.6 Soit $G = (S, A)$ un graphe et $S = \{s_1, \dots, s_n\}$ une énumération sans répétition des sommets de G . La matrice $M_G = (m_{ij})_{1 \leq i, j \leq n} \in \mathcal{M}_n(\mathbb{R})$ définie par

$$\forall i, j \in \llbracket 1, n \rrbracket, m_{ij} = \begin{cases} 1 & \text{si } s_i \rightarrow s_j \\ 0 & \text{sinon} \end{cases}$$

est appelée une **matrice d'adjacence** de G .

■ **Remarque 11.5** Il s'agit bien d'**une** matrice d'adjacence et pas de **la** matrice car elle dépend de l'énumération des sommets.

On en déduit ainsi une implémentation directe d'un graphe en représentant la matrice d'adjacence comme l'a fait pour des images.

Cette représentation permet d'accéder, en lecture comme en écriture, à une arête en temps constant. Cependant, pour récupérer les voisins d'un sommet, il est nécessaire de parcourir toute la ligne correspondante, donc en $O(|S|)$.

■ **Remarque 11.6** Dans la majorité des cas, A est petit par rapport à $|S|^2$. Ainsi, la matrice contient très peu de 1 et beaucoup de 0, on dit qu'elle est *creuse*. Une telle matrice *creuse* peut-être efficacement représenté par l'ensemble fini des arêtes (i, j) . Par exemple, avec une table de hachage.

1.5.ii Listes d'adjacences

La donnée de $v_+(x)$ pour chaque sommet $x \in S$ suffit à reconstruire A . On en déduit ainsi une représentation d'un graphe où on place dans un tableau chaque $v_+(x)$ représenté par une liste chaînée. L'accès en lecture ou en écriture à une arête est alors en $O(|A|)$ mais on peut parcourir les voisins sortant en $O(|A|)$ également. Pour un sommet donné, on peut même préciser $O(d_+(x))$.

Pour obtenir les voisins entrants, il est par contre nécessaire de tester la présence de x dans chacune des autres listes, on obtient donc un algorithme en $O(|S| + |A|)$: on parcourt chaque case du tableau des listes puis chaque maillon de listes d'adjacence.

1.5.iii Comparaison

opération	Matrice	Listes
complexité spatiale	$O(S ^2)$	$O(S + A)$
arête test	$O(1)$	$O(A)$
arête ajout	$O(1)$	$O(1)$
arête suppression	$O(1)$	$O(A)$

opération	Matrice	Listes
sommet ajout	$O(S ^2)$	$O(S)$
sommet suppression	$O(S ^2)$	$O(S)$
voisins/degré +	$O(S)$	$O(d_+(x))$
voisins/degré -	$O(S)$	$O(S + A)$

II Graphes non orientés

II.1 Définition et adaptation du vocabulaire

Définition II.1 Un graphe non orienté est un couple $G = (S, A)$ où S est un ensemble fini et A est un ensemble de paires d'éléments de S , c'est-à-dire d'ensembles à deux éléments $\{x, y\}$ où $x, y \in S$.

Les éléments de S sont appelés les **sommets** du graphe G et les éléments de A les **arêtes**. Si $\{x, y\} \in A$, on note $x \sim y$.

■ **Remarque 11.7** Si $G = (S, A)$ est un graphe orienté, on peut en déduire deux graphes non orientés :

- un graphe par défaut $G^- = (S, A^-)$ où

$$\forall x, y \in S, x \sim y \iff (x \rightarrow y \wedge y \rightarrow x)$$

- un graphe par excès $G^+ = (S, A^+)$ où

$$\forall x, y \in S, x \sim y \iff (x \rightarrow y \vee y \rightarrow x)$$

On a également A symétrique $\iff G^+ = G^-$. Cela correspond à un graphe où chaque arête est dans les deux sens $x \leftrightarrow y$ et, donc, on peut oublier l'orientation.

On reprend directement l'essentiel du vocabulaire des graphes orientés symétriques avec des simplifications :

Définition II.2 Soit $G = (S, A)$ un graphe et $x \in S$.

- On appelle *voisins* de x les éléments de

$$v(x) = \{ y \in S \mid x \sim y \}$$

- On appelle *degré* de x l'entier $d(x) = |v(x)|$.

Théorème II.1 $\sum_{x \in S} d(x) = 2|A|$

On étend directement la notion de **chemin** mais il faut faire attention au fait que **simple** n'a pas le même sens entre un graphe non orienté symétrique et un graphe non orienté. En effet, on $x \rightarrow y \rightarrow x$ est simple pour un graphe orienté alors que $x \sim y \sim x$ ne l'est pas vu qu'il s'agit de la même arête.

II.2 Connexité

Définition II.3 On définit \sim^* comme étant la clôture réflexive et transitive de \sim .

Théorème II.2

- \sim^* est une relation d'équivalence dont les classes sont appelées les **composantes connexes** du graphe.
- $x \sim^* y \iff$ il existe un chemin de x à y .

Définition II.4 Un graphe n'ayant qu'une classe d'équivalence pour \sim^* est dit **connexe**. Cela signifie qu'il existe un chemin entre toute paire de sommets.

III Graphe acyclique connexe

Définition III.1 On appelle **arbre** un graphe acyclique connexe. Quand on a distingué un sommet, on parle d'**arbre enraciné**.

IV Parcours

IV.1 Principe

On reprend le principe des parcours d'arbres : un parcours de graphe est une méthode qui étant donné un sommet s de départ parcourt tous les sommets accessibles depuis s une fois et une seule. A chaque visite d'un nouveau sommet on effectue un traitement qui dépend du problème considéré.

Tous les parcours de graphe ont la même structure :

- On marque tous les sommets comme étant non visités
- On ajoute s dans un ensemble de sommets à visiter V
- Tant que V est non vide
 - ★ On extrait un sommet x de V
 - ★ Si x a été marqué comme visité, on finit l'itération
 - ★ On marque x comme visité
 - ★ On traite x
 - ★ On ajoute à V tous les successeurs de x

Notons qu'on peut aussi optimiser la taille de V en ajoutant uniquement les successeurs non visités. Mais il faudra tout de même garder le test pour savoir si un sommet extrait est visité, car on peut ajouter un successeur déjà présent dans V et il ne faut pas le visiter de et il ne faut pas le visiter deux fois.

Pour marquer visité ou non des sommets, on considère un `bool array` donc le test est en c.t. $\mathcal{O}(1)$ et la c.s. $\mathcal{O}(|S|)$.

Tout l'enjeu, et donc les différentes notions de parcours, provient de l'implémentation de l'ensemble V . Au maximum V contiendra tous les successeurs donc une c.s. en $\mathcal{O}(|A|)$. Si on considère qu'ajout/extraction sont en $\mathcal{O}(1)$, comme on effectue autant d'itérations que d'éléments ajoutés à V on a une c.t. en $\mathcal{O}(|A|)$.

Voici le code OCaml générique de tout parcours en considérant un graphe donné par listes d'adjacence :

```
let parcours traitement g s =
  let visites = Array.make (Array.length g) false in
  let a_visiter = cree_ensemble () in
  let rec ajoute_liste l = match l with
    [] -> () | t::q -> ajoute t a_visiter; ajoute_liste q
  in
  ajoute s a_visiter;
  while not (est_vide a_visiter) do
    let x = extraction a_visiter in
    if not (visites.(x)) then begin
```

OCaml

```

    visites.(x) <- true;
    traitement x;
    ajoute_liste g.(x)
end
done

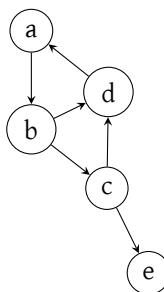
```

IV.2 Profondeur

Si on choisit d'implémenter l'ensemble V par une pile, on obtient un parcours appelé parcours en profondeur (*depth-first search*).

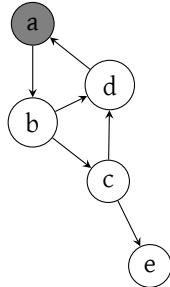
Rappel : les piles sont des structures LIFO (Last In First Out), i.e. dernier entré premier sorti.

Exemple de parcours en profondeur (on marque les sommets visités en gris)

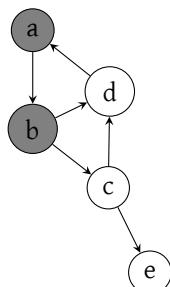


en partant de a :

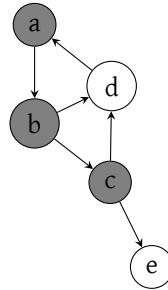
- Au départ $V = [a]$.
- Visite de a , $V = [b]$



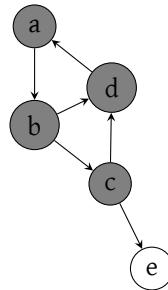
- Visite de b , $V = [c; d]$



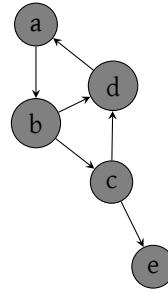
- . Visite de c , $V = [d; e; d]$



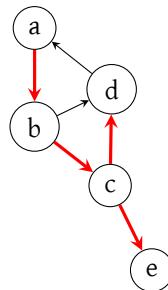
. Visite de d , $V = [a; e; d]$



- Extraction sans visite de a car déjà visité, $V = [e; d]$
- Visite de e , $V = [d]$



- Extraction sans visite de d car déjà visité, $V = []$
- En marquant les arêtes qui ont été empruntés



on remarque que ce parcours permet d'obtenir un arbre enraciné en a comprenant tous les sommets accessibles depuis a . L'ordre de visite des sommets (a, b, c, d, e) correspond alors à l'ordre de visite d'un parcours en profondeur de cet arbre.

Notons $t_i(x)$ le numéro de l'étape où le sommet x a été visité et $t_o(x)$ le numéro de l'étape où on a extrait le dernier descendant de x . C'est-à-dire que si la pile est π avant d'avoir traité x , on note $t_o(x)$ le numéro de l'étape où la pile redevient π .

x	$t_i(x)$	$t_o(x)$
a	2	8
b	3	8
c	4	7
d	5	6
e	7	7

Ici, on a . On remarque que ces temps sont bien parenthésés : si $t_i(x) <$

$t_i(y) \leq t_o(x)$ alors $t_o(x) > t_o(y)$ (comme si on considère t_i l'indice d'une parenthèse ouvrante et t_o l'indice d'une parenthèse fermante dans une expression bien parenthésée, d'où le nom). Si on pose $I(x) = [t_i(x); t_i(y)]$, on voit qu'étant donné deux sommets x et y , on a soit $I(x) \cap I(y) = \emptyset$ soit $I(x) \subset I(y)$ soit $I(x) \subset I(y)$. De plus, la relation de couverture associée à cet ordre est exactement l'arbre défini précédemment.

■ **Remarque 11.8** Pour programmer ce parcours en Caml, on a deux options :

- Utiliser directement le module Stack de OCaml avec le code précédent en posant :

```
OCaml let cree_ensemble () = Stack.create ()
let est_vide pi = Stack.is_empty pi
let ajoute x pi = Stack.push x pi
let extraction pi = Stack.pop pi
```

- Tirer partie de la pile naturelle des appels récursifs :

```
OCaml let parcours traitement g s =
  let visites = make_vect (vect_length g) false in
  let rec parcours_aux l =
    match l with
    | [] -> () (* Plus de sommets *)
    | x::q -> if not (visites.(x))
      then begin
        visites.(x) <- true;
        traitement x;
        parcours_aux (g.(x) @ q)
      end else parcours_aux q
  in parcours_aux [s]
```

Ici on remarque l'intérêt du test tardif pour savoir si on a déjà visité un sommet : on se contente de rajouter **tous** les voisins de x avec une concaténation.

Avec un filtre on peut tout de même ajouter moins de sommets, mais attention il faut tout de même préserver le test pour éviter le double traitement.

```

Ocaml
let parcours traitement g s =
  let visites = make_vect (vect_length g) false in
  let rec filtre l = match l with [] -> []
    | x::q -> if visites.(x) then filtre q else x::filtre q
  in
  let rec parcours_aux l =
    match l with
    | [] -> () (* Plus de sommets *)
    | x::q -> if not (visites.(x))
      then begin
        visites.(x) <- true;
        traitement x;
        parcours_aux (filtre g.(x) @ q)
      end else parcours_aux q
  in parcours_aux [s];;
```

Pour finir, une petite note sur la complexité temporelle de cette variante. On pourrait croire que l'ajout d'une concaténation augmente la complexité, mais ce n'est pas le cas la concaténation se substitue à l'ajout des successeurs un par un : on a donc remplacé n opérations en $\mathcal{O}(1)$ par une opération en $\mathcal{O}(n)$.

Quels sont les problèmes qu'on peut résoudre avec un parcours en profondeur ?

- L'accessibilité : les sommets marqués en fin de parcours sont exactement les sommets x tels que $s \rightsquigarrow x$.
- La forte connexité : posons $G^{-1} = (S, A^{-1})$ où $A^{-1} = \{ (v, u) \mid (u, v) \in A \}$ le graphe miroir de G obtenu en changeant l'orientation de toutes les arêtes. On a $x \rightsquigarrow y$ dans G ssi $y \rightsquigarrow x$ dans G^{-1} .

Si G est fortement connexe et $s \in G$, alors pour tout sommet x on a $s \rightsquigarrow x$ **et** $x \rightsquigarrow s$.

Réiproquement, s'il existe un sommet s satisfaisant cela, alors pour tout $x, y \in G$, on a un chemin $x \rightsquigarrow s \rightsquigarrow y$ et donc G est fortement connexe.

On en déduit un test en deux parcours en profondeur de la forte connexité : un premier parcours sur G et un second sur G^{-1} .

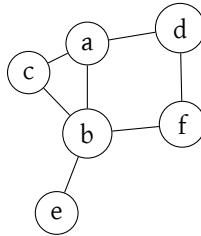
- La recherche d'un cycle : si on ajoute un sommet à visiter alors qu'on est déjà en train de visiter ses successeurs, c'est qu'il y a un cycle contenant ce sommet.
- La recherche d'une solution dans un labyrinthe.
- La 2-connexité : ici on exploite une propriété des revisites du parcours en profondeur. Si on voit deux fois un sommets c'est qu'il existe deux chemins pour l'atteindre depuis s . On peut en déduire alors un algorithme de test de la 2-connexité : G est 2-connexe si en enlevant une arête quelconque G reste connexe.

IV.3 Largeur

Si on choisit d'implémenter l'ensemble V par une file, on obtient un parcours appelé parcours en largeur (*breadth-first search*).

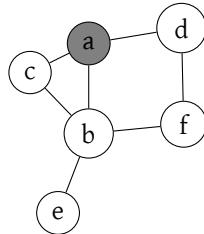
Rappel : les files sont des structures FIFO (First In First Out), i.e. premier entré premier sorti.

Exemple de parcours en largeur (on marque les sommets visités en gris et on omet les sommets déjà visités) du graphe

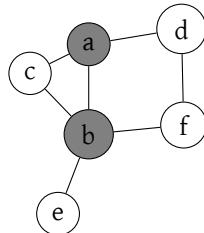


en partant de a :

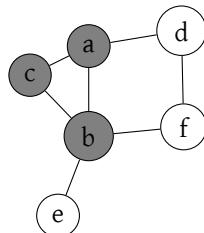
- Visite de a , $V = [b; c; d]$,



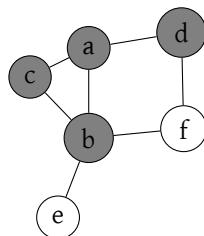
- Visite de b , $V = [c; d; a; e; f]$,



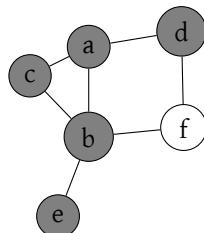
- Visite de c , $V = [d; a; e; f; a; b]$,



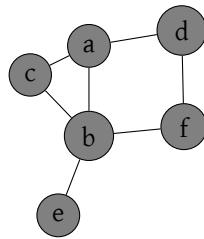
- Visite de d , $V = [e; f; a; b; a; f]$,



- Visite de e , $V = [f; a; b; a; f; b]$,



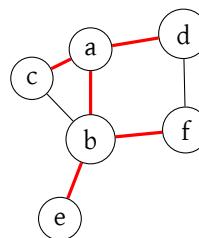
- Visite de f , $V = [a; b; a; f; b; b; d]$,



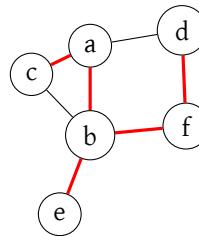
- Plus que des sommets visités dans V .

L'intérêt principal du parcours en largeur c'est qu'il garantit que chaque sommet soit découvert au plus tôt et donc il permet d'extraire des plus courts chemins depuis s .

Dans notre exemple, on obtient un arbre issu du parcours en largeur qui est



et tous les chemins de a vers les feuilles sont des plus courts chemins dans le graphe initial. Par opposition, l'arbre suivant issu d'un parcours en profondeur



ne vérifie pas cela, le chemin $a \rightarrow b \rightarrow f \rightarrow d$ n'est pas minimal.

La plupart des questions auxquels on a pu répondre par parcours en profondeur pourront être résolues par parcours en largeur. Le problème du parcours en largeur c'est qu'on doit contenir dans la file **tous les sommets** à un distance donné de s (on parle de niveau). Si on considère un arbre n -aire complet issu de s , le k -ième niveau contient exactement n^k sommets et donc la file du parcours en largeur croît de manière exponentielle. Avec un parcours en profondeur, on se contente d'empiler les descendants le long d'une branche de la racine à une feuille, ce qui est de l'ordre de kn quand on descend au niveau k .

■ **Remarque 11.9** Pour implémenter ce parcours en Caml, on utilise le type `Queue` des files, en posant :

Ocaml

```

let cree_ensemble () = Queue.create ()
let est_vide pi = Queue.is_empty pi;;
let ajoute x pi = Queue.add x pi;;
let extraction pi = Queue.take pi;;

```

V Chemin de plus petit poids

V.1 Graphe pondéré

On considère maintenant une pondération sur les arêtes d'un graphe. C'est-à-dire une fonction $w : A \rightarrow \mathbb{R}$. On parle de graphe pondéré et $w(a)$ est appelé le poids de l'arête a . Le poids d'un chemin est la somme des poids de ses arêtes.

On se pose alors la question suivante : étant donné x et y existe-t-il un chemin de plus petit poids de x à y , et si oui comment l'obtenir ?

Notons qu'il est possible de trouver un cycle dont la somme des poids soit < 0 , en sorte qu'en prenant ce cycle plusieurs fois on puisse rendre le poids d'un chemin aussi petit qu'on le souhaite. C'est en ce sens qu'il n'existe pas forcément de chemin de plus petit poids fini.

En donnant à chaque arête le même poids 1, on retrouve naturellement la notion de plus court chemin dans un graphe.

V.2 Dijkstra : source unique

On considère ici des poids dans \mathbb{R}_+^* , ainsi prolonger un chemin a forcément un coût.

On va construire un sous-graphe $G' \subset G$ et un tableau d tel que d_x contienne

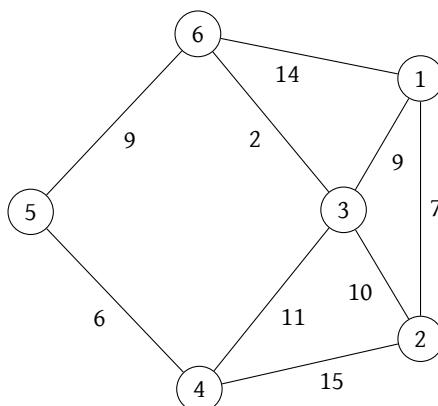
- la distance minimale entre s et x pour tout $x \in G'$.
 - la distance minimale entre s et x passant par des sommets de G' .

On commence avec $G' = \emptyset$. $d.s = 0$ et $d.x = \infty$ pour $x \neq s$.

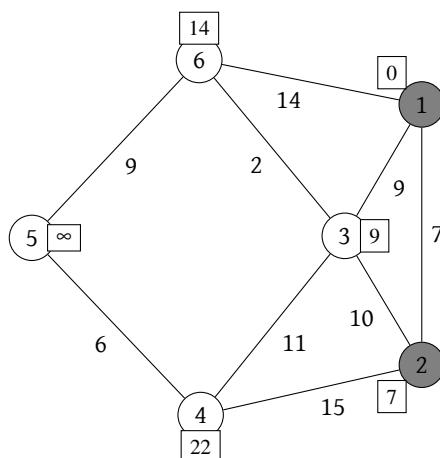
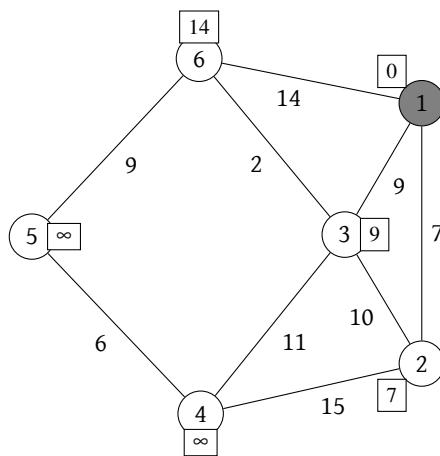
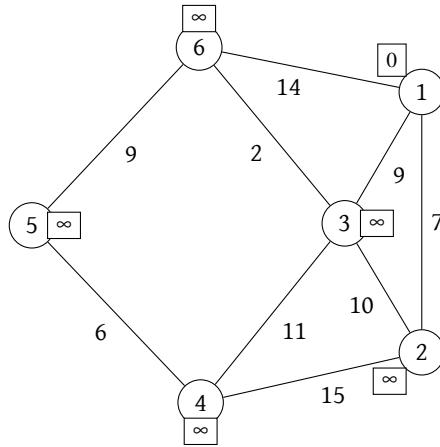
Si on a construit G' , on cherche un sommet qui n'est pas dans G' et qui a le d_x minimal. On le rajoute dans G' et on met à jour ses voisins pour que d contienne toujours les longueurs passant par $G' : d_y = d_x + w(x, y)$.

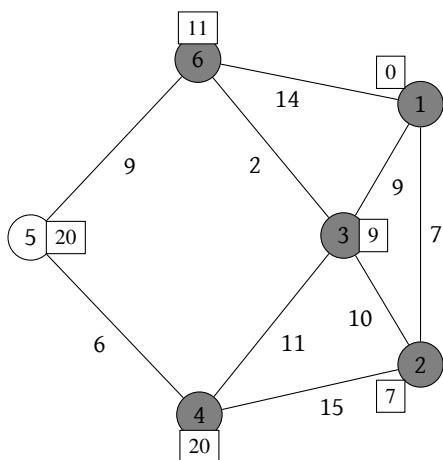
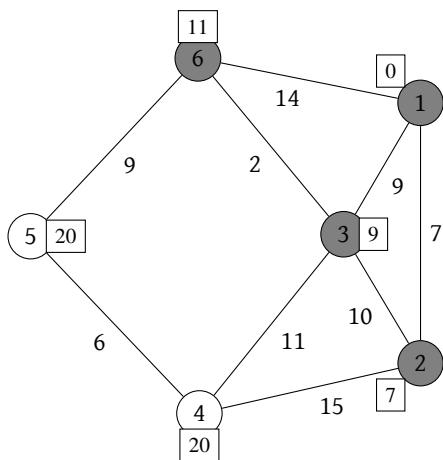
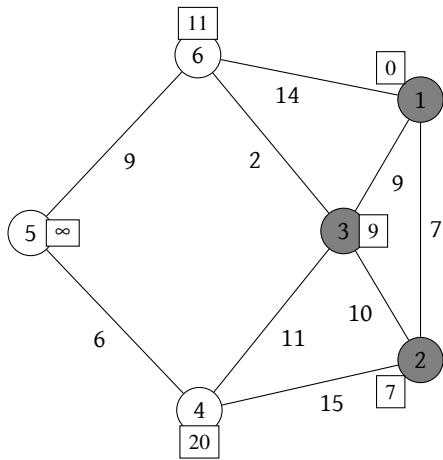
On itère jusqu'à avoir ajouté tous les sommets accessibles depuis s .

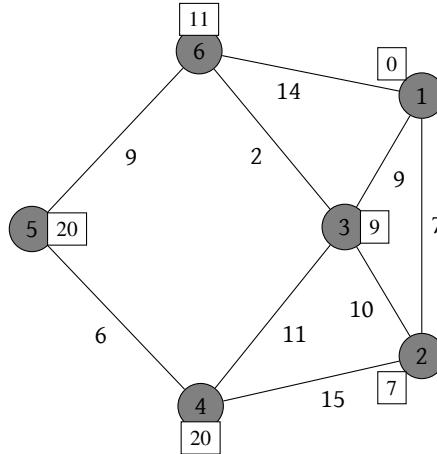
On va le faire sur l'exemple suivant :



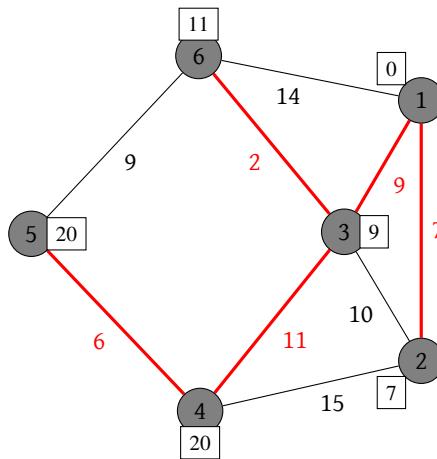
Les sommets de G' seront marqués en gris et les valeurs de d indiquées à coté des noeuds dans des carrés.







On peut indiquer en gras les arêtes qui ont permis d'améliorer les valeurs du tableau d :



Les chemins de plus petits poids sont exactement ceux qui empruntent ces arêtes. Pour permettre de calculer efficacement ces chemins, il suffit de maintenir un second tableau p tel que si on a ajouté une arête (u, v) alors on a $p.v = u$.

Théorème V.1 Cet algorithme permet d'obtenir les poids des chemins minimaux issus de s .

Démonstration.

Pour cela il faut montrer que lorsqu'on considère un sommet à ajouter dans G' , on a bien la distance minimale. Si ce n'est pas le cas en ajoutant x , alors cela signifie qu'il existe un chemin plus court de s à x . Soit y le dernier sommet de G' dans ce chemin. On a forcément y suivi d'un sommet $z \neq x$. Sinon on aurait l'optimalité. Mais $d_x \leq d_z = d_y + w(y, z)$ et donc le chemin ne peut être plus petit que d_x . ■

Pour implémenter cet algorithme, on considère une structure de file de priorité dont on rappelle l'interface :

- `cree_file ()`
- `ajoute x p f` qui ajoute x dans la file f avec la priorité p
- `min_priorite f` qui renvoie la priorité minimale d'un élément de f (utile pour la condition d'arrêt de Dijkstra quand le graphe n'est pas connexe)
- `extrait_minimum f` qui renvoie l'élément de priorité minimale et le supprime de la file

- `est_vide f`
- et une fonction spécifique à Dijkstra `diminue_priorite x p f` qui remplace la priorité de `x` dans `f` par `p`.

Avec une structure de tas sur des couples (*priorité, valeur*), on peut ajouter, extraire et diminuer en c.t. $\mathcal{O}(\log n)$ (Attention, pour pouvoir diminuer la priorité, il suffit de changer la priorité dans le tas et faire remonter le sommet le long d'un chemin vers la racine, mais pour que la complexité soit bonne, il faut pouvoir accéder en $\mathcal{O}(1)$ à la case correspondant à une valeur, donc maintenir une table de correspondance).

En OCaml, on va représenter un graphe pondéré par des listes d'adjacence pondérées, c'est-à-dire que pour chaque arête (u, v) on aura dans la liste de u un couple $(v, w(u, v))$.

Avec cette structure de file de priorité, on obtient le code OCaml suivant :

```
let infini = max_int
let dijkstra g s =
  let n = vect_length g in
  let pred = make_vect n (-1) in
  let dists = make_vect n infini in
  let a_visiter = cree_file () in
  dists.(s) <- 0;
  for i = 0 to n-1 do
    ajoute i dists.(i) a_visiter;
  done;
  let rec traite_liste x l =
    match l with
    | [] -> ()
    | (y,poids)::q -> let nd = dists.(x) + poids in
      if nd < dists.(y)
      then begin
        dists.(y) <- nd;
        pred.(y) <- x;
        diminue_priorite y nd a_visiter
      end
    in
    while not (est_vide a_visiter) && min_priorite a_visiter < infini do
      let x = extrait_minimum a_visiter in
      traite_liste x g.(x)
    done;
  dists, pred
```

OCaml

Au départ, tous les sommets sont dans la file et à chaque étape on extrait un sommet. Il n'est donc pas nécessaire de maintenir une marque sur les sommets visités. La condition d'arrêt n'est pas uniquement lié au fait que la file soit vide car si le graphe est non connexe, des sommets avec distance ∞ ont été ajoutés dans la file au départ. Mais quand on extrait un sommet avec une priorité ∞ c'est nécessairement qu'il n'est pas accessible depuis le s.

La complexité spatiale de cet algorithme est celle de la file de priorité donc $\mathcal{O}(|S|)$. Pour la complexité temporelle, on a au départ la création de la file en $\mathcal{O}(|S| \log |S|)$ puis $|S|$ itérations de la boucle principale avec une extraction du minimum en $\mathcal{O}(\log |S|)$ et au maximum $|A|$ itérations de parcours d'arêtes ayant au maximum $\mathcal{O}(\log |S|)$ en raison de diminuer minimum. Au total, on a donc une complexité temporelle en $\mathcal{O}((|S| + |A|) \log |S|)$.

V.3 TODO Floyd-Warshall : toutes paires

On considère maintenant des poids quelconques.

Soit M^k la matrice telle que $M_{i,j}^k$ contient le poids du plus court chemin de i à j n'utilisant que des sommets intermédiaires de 1 à k (on place ∞ s'il n'y en a pas).

On pose donc $M_{i,j}^0 = w(i, j)$.

On a $M_{i,j}^k = \min(M_{i,j}^{k-1}, M_{i,k}^{k-1} + M_{k,j}^{k-1})$.

On peut faire ce calcul en place (boucles sur k puis i, j).

Complexité en temps $O(|S|^3)$ et en espace $O(|E|^2)$.

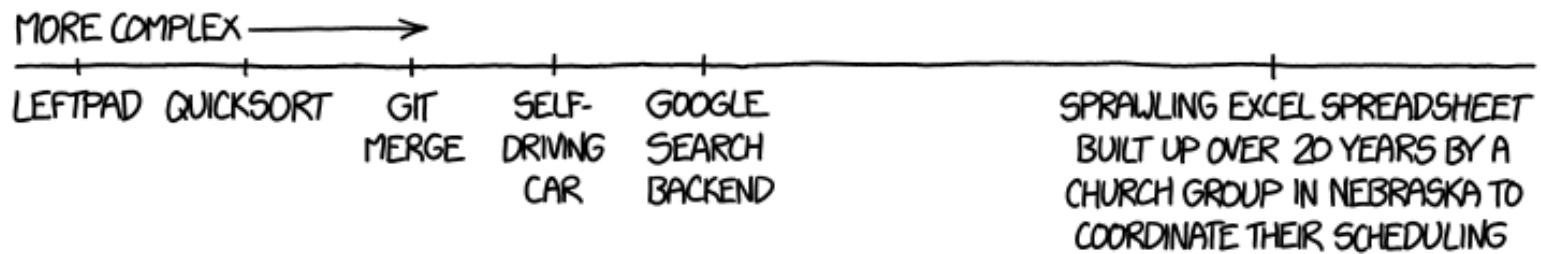
Détection de cycle négatif : $M_{i,j} < 0$.

IV

Algorithmique

12	Introduction à l'analyse des algorithmes	165
I	État d'un programme	
II	Terminaison	
III	Correction	
IV	Complexité	
V	Exercices	
13	Complexité amortie	187
I	Introduction	
II	Techniques de calcul	
14	Recherche par Force brute	197
I	Principe	
II	Recherche par retour sur trace (backtracking)	
III	Stratégies d'énumération	
IV	Droite de balayage	
15	Algorithmes gloutons	219
I	Principe	
II	Construction de l'arbre de Huffman	
III	Sélection d'activités	
IV	Principe général des preuves d'optimalité	
V	Ordonnancement de tâches	
VI	Exercices	
16	Division en sous-problèmes	237
I	Diviser pour régner	
II	Meet in the middle	
III	Dichotomie pour passer de décision à optimisation	
IV	Problèmes	
17	Algorithmique des textes	251
I	Recherche dans un texte	
II	Compression	
III	Problèmes supplémentaires	

ALGORITHMS BY COMPLEXITY



12. Introduction à l'analyse des algorithmes

Source de l'image d'en-tête XKCD #1667

■ Note 12.1 Roadmap :

- reprendre les exemples dans les trois langages.
- rajouter plus exercices.

■ **Remarque 12.1** Ce chapitre présente les trois grands principes qui nous serviront de guide pour analyser les algorithmes et les programmes :

- La **terminaison** : l'algorithme termine-t-il au bout d'un nombre fini d'étapes quelle que soit l'entrée ?
- La **correction** : le résultat rendu est-il celui qui était attendu ?
- La **complexité** : combien de temps prend le programme selon la taille de l'entrée ? Combien d'espace mémoire occupe-t-il ?

Savoir répondre à ces questions, c'est pouvoir prédire, avant d'avoir écrit la moindre ligne de code, ce qui va se passer.

■ État d'un programme

Avant de commencer à raisonner sur les algorithmes, il est nécessaire de préciser la notion d'état qui correspond à un instantané de l'environnement d'exécution d'un programme lorsque son exécution est interrompue. Bien entendu, la description complète d'un tel état ne serait pas forcément pertinent, car cela prendrait en compte l'ensemble de la mémoire. Le plus souvent, on considère uniquement ce qui est important pour ce qu'on étudie.

Ainsi, si on considère le programme suivant :

```

1 int a = 3;
2 int b = 2;
3
4 a = b;

```

L'état du programme à l'entrée de la ligne 4 est

variable	valeur
a	3
b	2

et l'état à la sortie de la ligne 4 est

a	2
b	2

Parfois, on aura besoin de plus d'information dans l'état comme la position en mémoire de certaines données. Mais assez souvent, pour les algorithmes qui nous intéressent, on pourra adopter un point de vue assez abstrait de l'état d'un algorithme comme étant une fonction partielle des noms vers les valeurs.

Pour la terminaison et la correction, on va considérer des propriétés logiques dépendant de l'état. Par exemple $a \geq 0 \wedge b \geq a$.

II Terminaison

Définition II.1 On dit qu'un algorithme **termine** quand il n'exécute qu'un nombre fini d'étapes sur toute entrée.

■ **Remarque 12.2** Cela n'empêche pas que ce nombre d'étapes puisse être arbitrairement grand en fonction des entrées. ■

Un algorithme qui n'utilise ni boucles inconditionnelles ni récursivité termine toujours. Ainsi, la question de la terminaison n'est à considérer que dans ces deux cas.

Considérons par exemple l'algorithme suivant qui, étant donné un entier naturel n strictement positif, inférieur à 2^{30} , détermine le plus petit entier k tel que $n \leq 2^k$:

```

int plus_grande_puissance2(int n)
{
    int k = 0;
    int p = 1;

    while (p < n)
    {
        k = k+1;
        p = p*2;
    }

    return k;
}

```

On remarque que p prend successivement pour valeurs toutes les puissances de 2 jusqu'à une éventuelle sortie de boucle. Or, il existe une puissance de 2 supérieure ou égale à n , donc, une fois atteinte, la condition de la boucle while n'est plus remplie et l'algorithme termine.

■ **Remarque 12.3** Prouver la terminaison n'est pas une question facile, elle peut-même être insoluble. Par exemple, si on considère le programme suivant :

Python

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

être capable de prouver sa terminaison revient à prouver la conjecture de Collatz (encore dite de Syracuse).

On peut aussi considérer le tri suivant :

Python

```
def bogo(t):
    while not est_trie(t):
        melange(t)
```

Il semble très improbable que cet algorithme ne termine pas, d'ailleurs on peut prouver qu'il termine avec probabilité 1, mais on ne peut pas exclure le cas où il mélange indéfiniment. ■

II.1 Variant de boucle

Pour prouver la terminaison d'une boucle conditionnelle, on utilise un **variant** de boucle.

Définition II.2 Un **variant de boucle** est une quantité entière **positive** à l'entrée de chaque itération de la boucle et qui diminue **strictement** à chaque itération.

Dans l'exemple précédent, la quantité $n - p$ est un variant de boucle :

- Au départ, $n > 0$ et $p = 1$ donc $n - p \geq 0$
- Comme il s'agit d'une différence de deux entiers, c'est un entier. Et tant que la condition de boucle est vérifiée $p < n$ donc $n - p > 0$.
- Lorsqu'on passe d'une itération à la suivante, la quantité passe de $n-p$ à $n-2p$ or $2p-p > 0$ car $p \geq 1$. Il y a bien une stricte diminution.

■ **Remarque 12.4** Ici, en sortie de boucle, $n - p \leq 0$. On fait donc bien attention de préciser que la quantité est positive tant que la condition de boucle est vérifiée. ■

Si on a un variant de boucle qui vaut initialement n avant d'entrer dans la boucle, celle-ci effectue au plus n itérations car le variant diminue au moins de 1 à chaque étape.

Théorème II.1 Si une boucle admet un variant de boucle, elle termine.

II.2 Exemple de la recherche dichotomique

On considère ici la recherche dichotomique dans un tableau trié d'entiers. Étant donné un tableau t de taille $n > 0$ et un entier x dont on cherche à déterminer sa présence dans le tableau entre les indices i et j , on considère l'algorithme suivant :

- Si $i > j$, alors il n'y a pas de sous-tableau et on renvoie `false`.
- Sinon, soit m l'élément d'indice $p = \left\lfloor \frac{i+j}{2} \right\rfloor$.
 - * Si $x = m$, on renvoie `true`
 - * Si $x < m$, on continue la recherche dans le sous-tableau des indices i à $p - 1$.
 - * Si $x > m$, on continue la recherche dans le sous-tableau des indices $p + 1$ à j .

Le programme suivant présente une implémentation de cet algorithme en C.

```
int rech_dicho(int *t, size_t n, int x)
{
    /* renvoie un indice de x
       si x est dans le tableau et -1 sinon */
    int i = 0;
    int j = n-1;

    while (i <= j)
    {
        int p = (i+j)/2;
        int m = t[p];

        if (x == m)
        {
            return p;
        }
        else if (x < m)
        {
            j = p-1;
        }
        else
        {
            i = p+1;
        }
    }

    return -1;
}
```

■ **Remarque 12.5** On aurait pu écrire $i+(j-i)/2$ et non $(i+j)/2$ afin d'éviter des erreurs de dépassement dans le calcul de $i+j$. ■

Ici, la terminaison n'est pas immédiate, on va la prouver à l'aide d'un variant de boucle. On considère ainsi la quantité $d(i, j) = j - i$.

- Comme le tableau est non vide, $d(0, n - 1) \geq 0$. Ensuite, la condition de boucle est équivalente à $d(i, j) \geq 0$, donc cette quantité est bien entière et positive à l'entrée de chaque itération.
- Quand on passe à l'itération suivante, on passe
 - * soit de $d(i, j)$ à $d(i, p - 1)$. Or $d(i, p - 1) = p - 1 - i = \left\lfloor \frac{i+j}{2} \right\rfloor - 1 - i < \frac{i+j}{2} - i \leq$

$$j - i = d(i, j).$$

- * soit de $d(i, j)$ à $d(p + 1, j)$. Or $d(p + 1, j) = j - p - 1 = j - \left\lfloor \frac{i+j}{2} \right\rfloor - 1 < j - \frac{i+j}{2} \leq j - i = d(i, j)$.

Dans tous les cas, $d(i, j)$ diminue strictement.

Ainsi, il s'agit d'un variant de boucle et l'algorithme termine.

■ **Remarque 12.6** On remarque que le programme récursif suivant réalise également cet algorithme.



!langspec(ocaml)(c)(python)

Il suffit alors d'appeler `rech_dicho t 0 (Array.length t - 1) x` pour faire une recherche sur tout le tableau.

Il suffit alors d'appeler `rech_dicho(t, 0, len(t)-1, x)` pour faire une recherche sur tout le tableau.

Ici, il n'y a pas explicitement de boucle, mais le même principe peut être mis en place pour prouver que le nombre d'appels récursifs est majoré, et donc que toute exécution termine. En effet, la quantité $d(i, j) = j - i$ diminue pour les mêmes raisons à chaque appel récursif et reste entière positive.

II.3 Exemple du tri à bulle

On considère le tri à bulles dans une implémentation naïve. On effectue ainsi une série de balayages d'un tableau : on parcourt le tableau de gauche à droite et si deux éléments consécutifs sont dans le désordre, on les permute. À l'issu d'un tel balayage, si on a effectué au moins un échange, on recommence, sinon on s'arrête, car c'est le signe que le tableau est trié qu'on le prouvera à la fin de ce paragraphe.

On obtient alors le code suivant :

```
void swap(int t[], int i, int j)
{
    int temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}

void tri_bulles(int t[], int nb)
{
    bool echange = true;
    while(echange)
    {
        echange = false;
        for (int i = 0; i < nb-1; i++)
        {
            if (t[i] > t[i+1])
```

```
        {  
            echange = true;  
            swap(t, i, i+1);  
        }  
    }  
}
```

Si la question de la terminaison de la boucle `for` ne se pose pas, celle de la boucle `while` mérite de s'y attarder pour deux raisons :

- premièrement pour la terminaison elle-même
 - deuxième pour savoir si on peut majorer le nombre d'itérations, une question qu'on reverra avec la question de la complexité.

On va prouver la terminaison pour un type d'algorithme de tri qui généralise le tri à bulle. Pour cela, on commence par définir la notion d'inversion pour un tableau t de $|t|$ éléments :

Définition II.3 On dit qu'un couple $(i, j) \in \llbracket 0, |t| - 1, \rrbracket^2$ est une **inversion** pour t si $i < j$ et $t[i] > t[j]$.

On note $\text{Inv}(t)$ l'ensemble des inversions de t et $\text{inv}(t) = |\text{Inv}(t)|$ le nombre d'inversions.

On dira qu'une inversion de la forme $(i, i + 1)$ est une **inversion directe**.

On considère donc un algorithme qui résout une inversion directe dès qu'il y en a au moins une.

```
def tri(t):
    while existe_inversion_directe(t):
        i = debut_d_une_inversion_directe(t)
        echange(t, i, i+1)
```

Le tri à bulles est alors une manière de réaliser l'algorithme précédent en effectuant des inversions alors qu'on avance dans le tableau.

Théorème II.2 Pour un tableau t les propositions suivantes sont équivalentes :

1. t est trié dans l'ordre croissant
 2. t n'a pas d'inversions
 3. t n'a pas d'inversions directes

Démonstration.

- 1) \rightarrow 3) naturellement si t est trié dans l'ordre croissant, $i < i + 1$ entraîne $t[i] \leq t[i + 1]$, donc il ne peut y avoir d'inversions directes.
 - 2) \rightarrow 1) si $i < j$, comme t n'a pas d'inversions, on a $t[i] \not> t[j]$, c'est-à-dire $t[i] \leq t[j]$. Ainsi, le tableau est bien trié dans l'ordre croissant.
 - 3) \rightarrow 2) pour tout $i < |t| - 1$ on a $t[i] \leq t[i + 1]$. Soit $i < j$, on va raisonner par récurrence sur $j - i$ pour montrer que $t[i] \leq t[j]$.
 - ★ si $j - i = 1$, on n'a pas d'inversion directe donc $t[i] \leq t[j]$.
 - ★ si $j - i > 1$ et que $t[i] \leq t[j - 1]$, car $i < j - 1$ avec $j - 1 - i = (j - i) - 1$, alors $t[i] \leq t[j - 1] \leq t[j]$ car $(j - 1, j)$ n'est pas une inversion.

Ainsi, dans tous les cas on a bien $t[i] \leq t[j]$. Donc, t est trié par ordre croissant.

II.4 Lien avec la récursivité

II.5 Boucles imbriquées

III Correction

Pour parler de correction d'un algorithme, il est nécessaire d'identifier précisément ce qui doit être calculé par l'algorithme. Pour cela, on considère ici informellement des spécifications dépendant des entrées et du résultat de l'algorithme. On verra dans le chapitre sur la logique qu'il s'agit ici de prédictats logiques.

Voici des exemples de spécifications :

- le tableau t en sortie est trié dans l'ordre croissant
- la valeur renvoyée est le plus petit indice de x dans le tableau ou -1 s'il ne le contient pas.

Définition III.1 Un algorithme est **correct** vis-à-vis d'une spécification lorsque quelle que soit l'entrée

- il termine
- le résultat renvoyé vérifie la spécification.

On considère également la correction **partielle** en l'absence de terminaison :

Définition III.2 Un algorithme est **partiellement correct** vis-à-vis d'une spécification lorsque quelle que soit l'entrée le résultat renvoyé vérifie la spécification.

III.1 Invariant de boucle

Définition III.3 On considère ici une boucle (conditionnelle ou non).

Un prédictat est appelé un **invariant de boucle** lorsque

- il est vérifié avant d'entrer dans la boucle
- s'il est vérifié en entrée d'une itération, il est vérifié en sortie de celle-ci.

Quand la boucle termine, on déduit alors que l'invariant est vérifié en sortie de boucle. On cherche donc un invariant qui permette de garantir la spécification en sortie de boucle.

■ **Remarque 12.7** Pour les boucles inconditionnelles, il y a une gestion implicite de l'indice de boucle qui va se retrouver dans l'invariant. On peut alors considérer que la sortie de boucle s'effectue après être passé à l'indice suivant.

■

Dans le cas d'une boucle conditionnelle portant sur la condition P et ayant un invariant de boucle I , en sortie le prédictat $\neg P \wedge I$ (non P et I) sera vérifié.

On peut illustrer cela en reprenant la fonction `plus_grande_puissance2` vue à la partie Terminaison. On considère ici le prédictat $I(k, p) := 2^{k-1} < n$ et $p = 2^k$.

- En entrée de boucle, on a bien $2^{-1} < n$.
- Si le prédictat est vérifié en entrée d'itération. On a alors $2^{k-1} < n$ et comme on est entrée dans cette itération $p = 2^k < n$. Donc en sortie d'itération on aura bien $I(k + 1, 2p)$, car $2p = 2^{k+1}$.

Ainsi, ce prédictat est bien un invariant et en sortie de boucle (ce qui arrive nécessairement, car l'algorithme termine), le prédictat $I(k, p)$ signifie que $2^{k-1} < n$ et **la condition de sortie de boucle** qu'on a $n \leq 2^k$.

La valeur renvoyée est bien k tel que $2^{k-1} < n \leq 2^k$ ce qui était la spécification annoncée du programme.

III.2 Exemple du tri par sélection

Le programme suivant présente un algorithme de tri, appelé le *tri par sélection* dont on va analyser la complexité. Il s'agit d'un tri qui repose sur un principe simple, on va chercher le plus petit élément du tableau à trier et le placer à la position courante. On définit ainsi trois fonctions :

- `echange` réalise l'échange de valeurs entre deux cases du tableau
- `indice_minimum` renvoie l'indice de la plus petite valeur entre deux indices donnés
- `tri_par_selection` réalise le tri en parcourant le tableau du premier au dernier indice et en plaçant à la position courante le minimum restant.

```

void echange(int *tableau, int i, int j)
{
    int temp = tableau[i];
    tableau[i] = tableau[j];
    tableau[j] = temp;
}

void indice_minimum(int *tableau, int min_indice, int max_indice)
{
    int i = min_indice;

    for (int j = min_indice + 1; j <= max_indice; j++)
    {
        if (tableau[j] < tableau[i])
            i = j;
    }

    return i;
}

void tri_par_selection(int *tableau, int taille)
{
    for (int i = 0; i < taille; i++)
    {
        echange(tableau, i, indice_minimum(tableau, i, taille-1));
    }
}

```

Il n'y a pas de problèmes de terminaison ici, car toutes les boucles sont inconditionnelles. Pour prouver sa correction, on va considérer séparément les deux boucles.

- Boucle dans `indice_minimum`: on va valider l'invariant $I(i, j) := \forall k \in [i, j-1], \text{tableau}[i] \leq \text{tableau}[k]$.
 - ★ En entrée de boucle, on a $I(\min_indice, \min_indice + 1)$ vérifié directement.
 - ★ Si en entrée d'itération, $I(i, j)$ est vérifié, ce qui signifie que $\text{tableau}[i]$ est plus petit que les valeurs compris entre les indices i et $j - 1$. Alors, on distingue deux cas :
 - soit $\text{tableau}[j] < \text{tableau}[i]$ et alors en sortie i devient $i' = j$. On a alors $\text{tableau}[i'] = \text{tableau}[j] < \text{tableau}[i] \leq \text{tableau}[k]$ pour $k \in [1, j - 1]$. Donc $I(i', j + 1)$ est vérifié.
 - soit $\text{tableau}[i] \leq \text{tableau}[j]$ et ainsi on a pu prolonger le prédictat à $I(i, j + 1)$.

Ce prédictat est bien un invariant. Ainsi, en sortie de boucle, et donc avant de renvoyer sa valeur, on a bien $I(i, \text{taille})$ donc $\text{tableau}[i]$ est la plus petite valeur du tableau.

- Boucle dans `tri_par_selection`: on va valider l'invariant $T(i) :=$ le sous-tableau $\text{tableau}[0..i-1]$ des indices 0 à $i - 1$ est trié et ne contient que des valeurs plus petites que celles du sous-tableau $\text{tableau}[i..taille-1]$.

- * En entrée de boucle, le sous-tableau est vide donc trié.
- * Si en entrée d'itération, le prédictat est vérifié. On récupère l'indice j du minimum du sous-tableau $\text{tableau}[i..taille-1]$ à l'aide la fonction `indice_minimum`, par hypothèse $\text{tableau}[j]$ est alors supérieur ou égal à chaque élément de $\text{tableau}[0..i-1]$, en le plaçant à l'indice i , on a bien $\text{tableau}[0..i]$ qui est trié et par construction la valeur de $\text{tableau}[i]$ est inférieure à toutes celles de $\text{tableau}[i+1..taille-1]$. On a ainsi $T(i+1)$ vérifié en sortie d'itération.

Ce prédictat est bien un invariant. Ainsi, en sortie de boucle, $T(\text{taille})$ est vérifié : le tableau est trié.

IV Complexité

IV.1 Complexité dans le pire des cas

Considérons un algorithme pour lequel on peut associer à chaque entrée une notion de taille (par exemple le nombre d'éléments d'un tableau). Pour $n \in \mathbb{N}$, on note ainsi I_n l'ensemble des entrées de taille n pour cet algorithme. Pour une entrée e , on note $t(e)$ le temps pris, par exemple en seconde, par l'algorithme sur l'entrée e . De même, on note $s(e)$ l'espace mémoire maximal, par exemple en octets, occupé par l'algorithme au cours de cette exécution **sans compter la taille des entrées**.

Définition IV.1 On appelle :

- **complexité temporelle dans le pire des cas**, la suite $(C_n^t)_{n \in \mathbb{N}}$ telle que pour tout $n \in \mathbb{N}$, $C_n^t = \max_{e \in I_n} t(e)$.
- **complexité spatiale dans le pire des cas**, la suite $(C_n^s)_{n \in \mathbb{N}}$ telle que pour tout $n \in \mathbb{N}$, $C_n^s = \max_{e \in I_n} s(e)$.

Comme on va le voir, calculer explicitement ces suites n'a pas beaucoup d'intérêt tant elles sont dépendantes de la manière dont on mesure le temps et l'espace. Ce qui compte ici, c'est de connaître l'*ordre de grandeur* de ces complexités en fonction de n .

Pour un tableau de taille n , ce programme va effectuer n itérations et sa complexité est ainsi de l'ordre de n . Il est possible d'être très précis en considérant les temps pris

- pour mettre en place l'appel de fonction et le passage des arguments
- par la gestion de l'indice de la boucle `for`
- pour la comparaison, puis pour l'affectation éventuelle
- pour mettre en place la valeur de retour afin que le résultat soit lu

On peut remarquer que la notion de pire cas dépend de la précision à laquelle on se place. Ici, si on ne s'intéresse qu'à l'ordre de grandeur, tous les tableaux de taille n sont équivalents. Par contre, si on cherche avec précision le pire cas, il est atteint avec un tableau trié par ordre croissant, car c'est le cas qui effectue une affectation à chaque itération.

```
int maximum(int *tableau, int taille)
{
    int M = INT_MIN;
    for(int i=0; i<taille; i++)
    {
        if (M > tableau[i])
            M = tableau[i];
    }
    return M;
}
```

IV.2 Comparer des complexités

Avant de pouvoir comparer les complexités des algorithmes ou des programmes, il est nécessaire de mettre en place des outils pour en parler à la fois avec précision mais également sans rentrer dans des détails d'implémentation non pertinents.

En effet, comparons les deux fonctions suivantes permettant de chercher un élément dans un tableau.

```
int recherche(int *tab, int nb,
             int elem)
{
    for(int i=0; i<nb; i++)
    {
        if (elem == tab[i])
            return i;
    }
    return -1;
}
```

```
int recherche(int *tab, int nb,
             int elem)
{
    int ind = -1;

    for(int i=0; i<nb; i++)
    {
        if (elem == tab[i])
            ind = i;
    }
    return ind;
}
```

La fonction de droite semble moins efficace que celle de gauche, car la seconde sort tout de suite de la fonction dès qu'on a trouvé l'élément alors que la première continue à parcourir t .

Mais on doit se poser la question de la pertinence de cette optimisation selon le pire des cas. Ici, le pire des cas correspond à ne pas avoir `elem` dans le tableau, et à ce moment-là les deux fonctions fonctionnent de la même manière.

■ **Remarque 12.8** En fait, la fonction de droite sera souvent à privilégier, car il est souvent plus facile d'interrompre le flot que de faire en sorte de préserver l'état à la place du `return` pour pouvoir renvoyer la bonne valeur à la fin de la fonction.

De la même manière, il faut déterminer ce que l'on souhaite compter précisément :

- si on s'intéresse au temps mis, certaines opérations prennent moins de temps que d'autre (par exemple une addition par rapport à une multiplication) mais est-ce vraiment important à l'échelle considérée ?
- si on s'intéresse à l'espace mémoire, doit-on considérer la taille précise en octets ou se contenter d'une estimation plus grossière ?

Mis à part dans certains cadres assez spécifiques, on se contente le plus souvent d'un ordre de grandeur pour ces complexités. Pour cela, on utilise des relations de comparaisons de suites et une échelle de grandeur usuelle pour les comparer.

IV.2.i La notation grand O

Définition IV.2 Soit $(u_n)_{n \in \mathbb{N}}$ et $(v_n)_{n \in \mathbb{N}}$ deux suites de nombres réels non nuls, on dit que la suite $(u_n)_n$ est dominée par $(v_n)_n$ lorsque la suite quotient $\left(\frac{u_n}{v_n}\right)_n$ est bornée.

On note alors $u_n = O(v_n)$.

Cette dernière notation se lit u_n est un grand O de v_n .

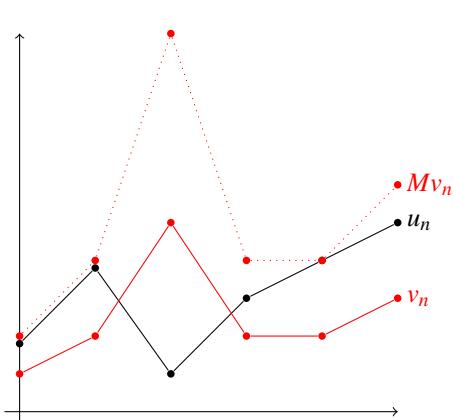
■ **Remarque 12.9** C'est bien cette locution qu'il faut avoir en tête quand on pense aux grands

O et il faut faire attention de ne pas considérer l'égalité en tant que telle sans s'assurer que ce l'on fait est licite. Quand on écrira par la suite $O(v_n)$ on signifiera *n'importe quelle suite qui soit un $O(v_n)$* .

Si $u_n = O(v_n)$, cela signifie qu'il existe un facteur $M > 0$ tel que pour tout entier n , on ait $-M|v_n| \leq u_n \leq M|v_n|$. Les variations de la suite $(u_n)_n$ sont ainsi entièrement contrôlées par les variations de $(v_n)_n$.

En informatique, on ne considère pour la complexité que des suites positives, ce qui permet de simplifier la relation : si $(u_n)_n, (v_n)_n$ sont des suites de réels strictement positifs, alors $u_n = O(v_n) \iff \exists M > 0, \forall n \in \mathbb{N}, u_n \leq M v_n$. C'est le cadre dans lequel on se place implicitement dans la suite de ce document.

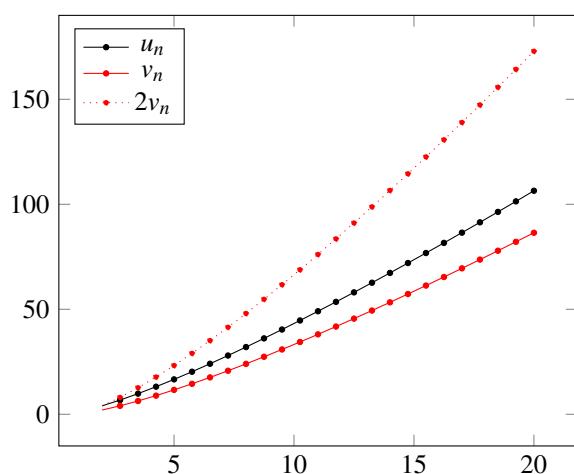
On peut visualiser graphiquement cette relation :



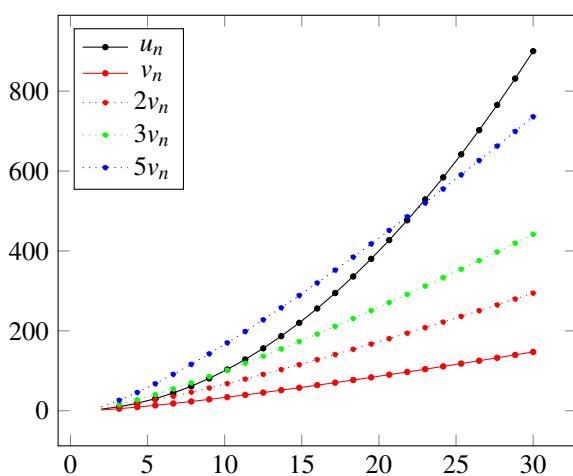
On a $u_n = O(v_n)$ si et seulement s'il est possible de multiplier les ordonnées de chaque point (n, v_n) par une constante afin que ces nouveaux points soient tous au-dessus des points (n, u_n) . On peut voir que la courbe déduite des v_n enveloppe, à un facteur près, celle des u_n .

Remarque : On a relié ici les valeurs des suites pour mieux mettre en valeur cette notion d'enveloppe.

Cette relation est une notion **asymptotique** : elle n'a d'intérêt que lorsqu'on considère des rangs au voisinage de l'infini. En effet, pour un nombre fini de termes, il est toujours possible de trouver un tel M , mais pour un nombre infini, ce n'est pas le cas.



Ici, on compare asymptotiquement les suites $(u_n)_n$ et $(v_n)_n$ où pour $n \in \mathbb{N}$, $u_n = n + n \log_2 n$ et $v_n = n \log_2 n$. Pour simplifier la visualisation, on a tracé les fonctions correspondantes. On remarque qu'on a bien $n + n \log_2 n = O(n \log_2 n)$.



Par contre, si on compare les suites $(u_n)_n$ et $(v_n)_n$ où pour $n \in \mathbb{N}$, $u_n = n^2$ et $v_n = n \log_2 n$, on remarque que quelle que soit la valeur choisie pour M , il y aura un rang à partir duquel $u_n > Mv_n$.
Ici, $n^2 \neq O(n \log_2 n)$.

■ Remarque 12.10 On a ici utilisé le logarithme en base 2, noté \log_2 , qui est essentiel informatique : si $x = \log_2(n)$ alors $n = 2^x$ où x est un réel. On considère aussi $p = \lceil \log_2(n) \rceil$ qui est le plus petit entier égal ou supérieur à $\log_2(n)$. On parle de **partie entière supérieure** et on a alors $2^{p-1} < n \leq 2^p$. Cet entier p correspond alors au plus petit nombre de chiffre nécessaire pour pouvoir écrire n en binaire. On a $\lceil \log_2(n) \rceil = O(\log_2(n))$ et ainsi, le plus souvent, on ne considère pas la partie entière explicitement. De la même manière, $\log_2(n) = \frac{\ln n}{\ln 2} = O(\ln n)$. ■

Un cas important de grand O est celui des $O(1)$. Si $u_n = O(1)$, cela signifie que $(u_n)_{n \in \mathbb{N}}$ est une suite bornée.

IV.2.ii Échelle de comparaison

On rappelle les limites obtenues en mathématiques que l'on nomme **croissances comparées** :

$$\forall \alpha, \beta > 0, \lim_{n \rightarrow +\infty} \frac{(\ln n)^\alpha}{n^\beta} = 0$$

$$\forall \alpha \in \mathbb{R}, \forall \beta > 1, \lim_{n \rightarrow +\infty} \frac{n^\alpha}{\beta^n} = 0$$

Or, si $\frac{u_n}{v_n} \xrightarrow[n \rightarrow +\infty]{} 0$ a fortiori le quotient est borné et $u_n = O(v_n)$. Ainsi, on a les relations suivantes :

$$\forall \alpha, \beta > 0, (\log_2 n)^\alpha = O(n^\beta)$$

$$\forall \alpha \in \mathbb{R}, \forall \beta > 1, n^\alpha = O(\beta^n)$$

De plus, si $\alpha \geq \beta > 0$, $n^\beta = O(n^\alpha)$, $(\log_2 n)^\beta = O((\log_2 n)^\alpha)$ et $\beta^n = O(\alpha^n)$.

On se ramène souvent à des complexités qui sont des grand O de produits de ces suites.

IV.2.iii Ordre de grandeur et relation Θ

On vient de voir que $\log_2 n = O(n)$, mais on a également $\log_2 = O(n^2)$. Quand on cherche à caractériser la complexité par un grand O , on va souvent chercher le grand O le plus proche de la suite.

Il est possible de définir cela précisément en considérant des suites qui sont chacune des grand O l'une de l'autre.

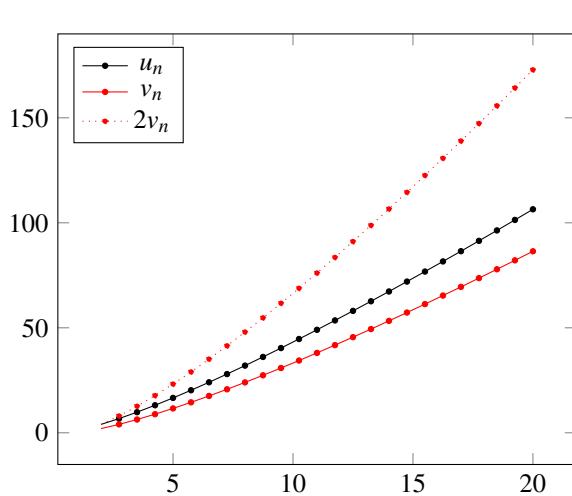
Par exemple, on a vu que $n \log_2 n + n = O(n \log_2 n)$, mais on a également $n \log_2 n = O(n + n \log_2 n)$.

Quand $u_n = O(v_n)$ et $v_n = O(u_n)$, on note $u_n = \Theta(v_n)$ qui est une relation symétrique qui correspond à la notion *avoir le même ordre de grandeur*. Très souvent, lorsque l'on parle de complexité, on utilise des grand O quand, en fait, on exprime des Θ . Par exemple, l'accès à un élément dans un tableau est en $O(1)$ et il ne serait pas précis de dire que c'est en $O(n)$ **même si c'est parfaitement correct**.

On peut visualiser cette relation Θ en considérant qu'il existe ainsi $M, M' > 0$ tels que $u_n \leq Mv_n$ et $v_n \leq M'u_n$. Mais on a alors

$$1/M'v_n \leq u_n \leq Mv_n$$

Ainsi, $u_n = \Theta(v_n)$ signifie qu'on peut encadrer $(u_n)_n$ entre deux multiples de $(v_n)_n$.



En reprenant la figure précédente, on observe visuellement

$$n \log_2 n \leq n \log_2 n + n \leq 2n \log_2 n$$

Avoir $u_n = \Theta(v_n)$ signifie donc que u_n évolue entre deux guides suivant les variations de v_n .

IV.2.iv Opérations sur les grands O

Si $u_n = O(w_n)$ et $v_n = O(w_n)$ alors $u_n + v_n = O(w_n)$. Ainsi, des grand O de même ordre s'ajoutent.

■ **Remarque 12.11** Comme on l'a vu précédemment, un grand O n'est pas très précis, et il est possible que par ajout on puisse obtenir un meilleur grand O. Par exemple : $n = O(n)$ et $\log_2 n - n = O(n)$ mais $n + \log_2 n - n = \log_2 n = O(n)$. Comme on ne considère ici que des suites strictement positifs, ce phénomène de compensation n'aura pas lieu. ■

Si $u_n = O(v_n)$ et w_n est une autre suite de réels strictement positifs, alors $u_n w_n = O(v_n w_n)$. On en déduit ainsi un principe qui nous sera utile par la suite $nO(1) = O(n)$.

IV.3 Complexités en temps classiques

On parle ici de complexité par raccourci pour parler de complexité dans le pire des cas en temps.

IV.3.i Complexité constante

On dit qu'un algorithme a une complexité constante quand $C_n^t = O(1)$. Il existe ainsi une constante M telle que le temps pris par l'algorithme **sur une entrée quelconque** soit inférieur à M .

De nombreuses opérations sont en temps constant sur les structures de données usuelles. Parmi celles-ci, citons-en deux essentielles :

- accéder à une case d'indice quelconque dans un tableau
- accéder à la tête ou à la queue d'une liste chaînée

Les algorithmes ou opérations en temps constant jouent un rôle primordial dans l'analyse de la complexité d'algorithmes, comme on le verra dans la partie suivante, car elles permettent de se concentrer sur les répétitions de ces opérations pour déterminer la complexité : une boucle qui se répète n fois et n'effectue que des opérations en temps constant dans son corps sera de complexité $nO(1) = O(n)$.

IV.3.ii Complexité linéaire

On dit qu'un algorithme a une complexité linéaire quand $C_n^t = O(n)$.

Cette complexité correspond à un traitement de temps constant sur chaque élément d'une entrée de taille n . C'est le cas de la recherche d'un élément dans un tableau ou de la recherche de son maximum.

Pour la recherche linéaire d'un élément, correspondant par exemple au programme ci-dessous, le pire cas correspond à ne pas avoir x dans tableau ce qui oblige à effectuer toutes les itérations. On a bien une complexité temporelle en pire cas de $O(n)$.

```
int recherche(int *tableau, int taille, int x)
{
    /* renvoie le plus petit indice i tel que tableau[i] = x
       ou -1 si x n'est pas dans le tableau */
    for(int i = 0; i < taille; i++)
    {
        if (tableau[i] == x)
            return i;
    }

    return -1;
}
```

IV.3.iii Complexité quadratique, polynomiale

On dit qu'un algorithme a une complexité quadratique quand $C_n^t = O(n^2)$. Par extension, on dit qu'il a une complexité polynomiale quand il existe $k \in \mathbb{N}$ tel que $C_n^t = O(n^k)$. Par extension, on parle parfois de complexité polynomiale pour des complexité plus précise en $O(n^\alpha)$ où α est un réel strictement positif.

L'exemple classique d'un algorithme quadratique est celui dû à un double parcours d'un tableau. On reprend ici l'algorithme de tri par sélection vu dans la partie Exemple du tri par sélection.

Afin d'analyser sa complexité, on procède fonction par fonction pour un tableau de taille n :

- `echange` est en temps constant. $O(1)$
- `indice_minimum` réalise un parcours du tableau et effectue des opérations en temps constant à chaque étape. La complexité est donc linéaire. $O(n)$
- `tri_par_selection` réalise également un parcours du tableau mais à chaque étape, on appelle `indice_minimum` qui est en $O(n)$, la complexité est donc en $nO(n) = O(n^2)$: elle est quadratique.

IV.3.iv Complexité logarithmique

On dit qu'un algorithme a une complexité logarithmique quand $C_n^t = O(\log_2 n)$.

Pour illustrer cette complexité, on reprend l'algorithme de recherche dichotomique vu dans la partie Exemple de la recherche dichotomique.

Chaque opération effectuée étant en temps constant, la complexité de cet algorithme correspond au nombre d'itérations, soit ici au nombre d'appels récursifs.

Si on considère un sous-tableau de $n = j - i + 1$ éléments lors de l'appel, un appel récursif se fera nécessairement sur un sous-tableau de $\lfloor n/2 \rfloor$ éléments. Ainsi, si $2^{k-1} < n \leq 2^k$, l'algorithme effectue moins de k itérations. En passant au logarithme, on a donc $k - 1 < \log_2 n \leq k$. Donc, le nombre d'itérations est en $O(\log_2 n)$ et c'est ainsi la complexité de l'algorithme.

■ **Note 12.2** Esquisser dès maintenant le lien entre longueur d'une branche dans un arbre de décision et complexité logarithmique ? ■

IV.3.v Complexité quasi-linéaire

On dit qu'un algorithme a une complexité quasi-linéaire quand $C_n^t = O(n \log_2 n)$. C'est le cas de la plupart des algorithmes efficaces de tri de n éléments. On peut même montrer qu'il s'agit de la complexité optimale.

Comme de nombreux algorithmes commencent par effectuer un tri avant d'effectuer un traitement linéaire, on retrouve des algorithmes quasi-linéaires par simple utilisation de ce tri.

IV.3.vi Complexité exponentielle

On dit qu'un algorithme a une complexité exponentielle quand $C_n^t = O(a^n)$ pour $a > 0$.

Un exemple fondamental d'un tel algorithme est celui de l'énumération de données, par exemple pour chercher une solution par force brute. En effet, il y a 2^n entrées codées sur n bits et un algorithme cherchant une solution ainsi parmi ces entrées aura une complexité en $O(2^n)$.

IV.3.vii Estimation de l'impact des complexités sur le temps

Afin de mesurer l'impact d'une complexité, on va considérer un algorithme qui s'exécute en 1 seconde sur un entrée de taille n , et on va calculer combien de temps prendrait ce même algorithme sur une entrée de taille $10n$.

Pour simplifier, on considère à chaque fois que C_n^t correspond exactement à l'ordre du grand O.

Complexité	Temps pour 10n	Temps pour 100n
1	1s	1s
$\log_2 n$	1,003s	1,007s
n	10s	1m40s
$n \log_2 n$	14,7s	3m13s
n^2	1m40s	2h46m40s
2^n	10^{19} années	10^{289} années.

■ **Remarque 12.12** Pour déterminer ces valeurs, on a considéré une unité de mesure de 1000ms afin d'en déduire une valeur de n .

Ainsi, si $\log_2 n = 1000$ on a $n = 2^{1000}$. Bien sûr, ici, ce nombre 2^{1000} n'est pas réaliste. Dans un contexte de mémoire finie, une complexité logarithmique est identifiable à une complexité constante. Cela justifie la terminologie quasi-linéaire.

Si $n \log_2 n = 1000$ alors $n \approx 140,2$. Or, $1402 \log_2 1402 \approx 14700ms$.

Si $2^n = 1000$, alors $n \approx 10$. Or $2^{100} \approx 10^{30}$.

IV.4 Calculer des complexités

Deux principes fondamentaux pour calculer des complexités :

- Si on effectue deux passes successives chacune en $O(u_n)$ alors la complexité globale est en $O(u_n)$. Il ne s'agit que de reformuler l'addition des grandeurs. Quand on a deux passes de complexité différente, il suffit d'utiliser la plus grande complexité. Par exemple, un algorithme qui commence par un tri en $O(n \log_2 n)$ et qui effectue ensuite un traitement en $O(n)$ sera de complexité globale $O(n \log_2 n)$ car le traitement est également en $O(n \log_2 n)$.
- Si on effectue u_n itérations et que chaque itération est en $O(v_n)$ alors l'algorithme a une complexité de $O(u_n v_n)$. Cela permet de compter le nombre de boucles imbriquées et de se contenter de regarder ce qui se passe dans le corps des boucles.

IV.5 Complexité à plusieurs paramètres

Jusqu'ici on a considéré des entrées dépendant d'un unique paramètre n , mais il est possible d'avoir des données dépendant de plusieurs paramètres.

On adapte directement la notation des grands O : si $(u_{n,p})$ et $(v_{n,p})$ sont deux suites de réels non nuls dépendant de deux paramètres, on note toujours $u_{n,p} = O(v_{n,p})$ quand le quotient est borné.

IV.5.i Données multidimensionnelles

Le cas le plus usuel de complexité dépendant de plusieurs paramètres est celui des données multidimensionnelles comme une image.

Si on considère une opération effectuant un traitement en temps constant sur chaque pixel d'une image de $w \times h$ pixels, cette opération aura une complexité en $O(wh)$. On ne peut plus parler de complexité linéaire ou quadratique ici car cela dépend d'une éventuelle relation entre w et h : si on ne travaille que sur des images de taille $1 \times h$ alors la complexité est $O(h)$, mais on ne travaille que sur des images carrées, donc pour lesquelles $w = h$, la complexité est $O(h^2)$.

Plus généralement, si on considère des données organisées dans des tableaux imbriqués, on effectuera un traitement sur chaque donnée à l'aide de boucles imbriquées non conditionnelles. La complexité sera alors celle du corps de boucles multipliée par le produit du nombre d'itérations de chaque boucle.

IV.5.ii Compromis entre paramètres

Dans certains cas, en particulier pour les graphes, on peut effectuer des traitements successifs dont la complexité ne s'exprime pas en fonction du même paramètre. Imaginons par exemple un programme ayant la structure suivante :

```
for (int i = 0; i<n; i++)
{
    /* corps de boucle en O(1) */
}

for (int j = 0; j<p; j++)
{
    /* corps de boucle en O(1) */
}
```

La complexité de la première boucle est en $O(n)$ et celle de la deuxième en $O(p)$. La complexité globale est en $O(n + p)$ car $n \leq n + p$ et $p \leq n + p$.

IV.6 Complexité en moyenne

On reprend ici les notations de la partie Complexité dans le pire des cas.

Définition IV.3 Lorsque pour tout $n \in \mathbb{N}$, I_n est fini, on appelle :

- **complexité temporelle en moyenne** la suite $(C_n^{t,m}) = \frac{1}{|I_n|} \sum_{e \in I_n} t(e)$.
- **complexité spatiale en moyenne** la suite $(C_n^{s,m}) = \frac{1}{|I_n|} \sum_{e \in I_n} s(e)$.

On peut étendre cette définition à un cadre infini en considérant une distribution de probabilité sur I_n et T_n la variable aléatoire associée à t sur I_n . Si T_n est d'espérance finie, on pourra parler de complexité en moyenne pour la suite des $E(T_n)$. Concrètement, on considère alors une fonction $p_n : I_n \rightarrow [0, 1]$ telle que $\sum_{e \in I_n} p(e) = 1$ et, lorsque la somme est définie, on note ainsi

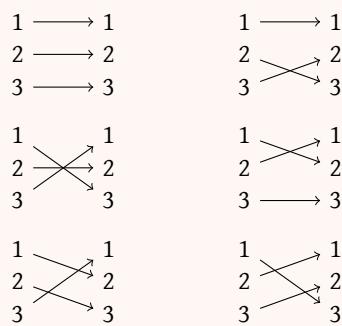
$$C_n^{t,m} = \sum_{e \in I_n} p(e)t(e)$$

$$C_n^{s,m} = \sum_{e \in I_n} p(e)s(e)$$

Un exemple usuel de calcul de complexité en moyenne est celui des tris. En effet, même si les entrées de taille n sont infinies, on peut considérer qu'un tableau de valeurs deux à deux distinctes est l'image par une permutation du tableau triée. Si le tableau est de taille n , on aura ainsi $n!$ permutations ce qui permet, du moment que l'algorithme de tri considéré ne dépend que cette permutation, de calculer la complexité en moyenne sur l'ensemble des permutations.

■ **Remarque 12.13** Les permutations d'un ensemble sont les applications bijectives de cet ensemble dans lui-même. Si l'ensemble contient n éléments, il y a $n!$ permutations.

Par exemple, les six permutations sur l'ensemble $\{1, 2, 3\}$ correspondent aux diagrammes sagittaires suivants :



Ces six permutations correspondant elles-mêmes, de gauche à droite et de haut en bas, aux tableaux $\{1, 2, 3\}$, $\{1, 3, 2\}$, $\{3, 2, 1\}$, $\{2, 1, 3\}$, $\{2, 3, 1\}$ et $\{3, 1, 2\}$.

IV.6.i Exemple de calcul de complexité temporelle en moyenne

On considère la recherche linéaire vue dans la partie Complexité linéaire. L'ensemble des entrées est ici infini, on va donc supposer pour faire le calcul qu'on ne considère que des tableaux de valeurs deux à deux distinctes et qu'on recherche un élément présent dans le tableau, chaque élément étant équiprobable.

Si on recherche le i -ème élément du tableau, l'algorithme effectue i itérations avant d'y accéder et de renvoyer son indice. Ainsi, le temps pour cet entrée est de iC où C est le coût d'une itération.

La complexité temporelle en moyenne est alors $C_n^{t,m} = \sum_{i=1}^n \frac{1}{n} iC = \frac{(n+1)C}{2} = O(n)$. On retrouve ici la même complexité que la complexité dans le pire des cas. La sortie prématuée de la boucle n'a donc aucune influence sur la complexité.

IV.7 Complexité amortie

Dans le cadre de l'étude des structures de données, il est fréquent de considérer non pas la complexité dans le pire des cas d'une opération mais celle d'une succession d'opérations divisée par le nombre d'opérations effectuées. Ainsi, on peut très bien avoir une opération ponctuellement plus coûteuse que les autres, mais en procédant ainsi on lisse le surcoût sur l'ensemble des opérations. On parle alors de **complexité amortie**.

L'étude de la complexité amortie est traitée dans le chapitre TODO.

IV.8 Pertinence de la complexité spatiale

Même si la complexité temporelle est le plus souvent celle qui est importante à calculer, certains algorithmes ont une complexité temporelle faible mais en contrepartie une complexité spatiale élevée. On parle alors de compromis temps-mémoire.

Un exemple classique d'un tel compromis est celui de la programmation dynamique où on passe d'une complexité temporelle exponentielle à une complexité temporelle polynomiale en stockant des valeurs intermédiaires pour ne pas les recalculer. En procédant ainsi, on passe d'une complexité spatiale constante à polynomiale.

Cela est illustré dans le programme suivant qui permet de déterminer le n -ième terme de la suite de Fibonacci, ce qui n'a pas d'intérêt informatique mais est caractéristique de récurrence que l'on résoudra par la programmation dynamique.



V Exercices

Exercice 12.1 On considère le programme suivant :

```
int multiplication(int x, int y, int c)
{
    int m = 0;
    while (y != 0)
    {
        m = m + x * (y % c);
        x = x * c;
        y = y / c;
    }
    return m;
}
```

On suppose que $c \geq 2$.

1. Montrer que cet algorithme termine .
2. Montrer que pour $x, y \in \mathbb{N}$, il renvoie le produit xy .
3. Réécrire ce programme en récursif. Peut-on déduire des preuves précédentes que l'algorithme termine et qu'il est correct.

■

Démonstration.

1. Comme $c \geq 2$, on a $y/c < y$ ainsi y est un variant de boucle.
2. On va commencer par noter x_0 et y_0 les valeurs initiales respectives de x et de y . On va également tenir compte du nombre d'itérations, ce qui revient à rajouter un compteur i au programme. On va également noter $y_0 = \sum_{j=0}^n a_j c^j$.
 - $y = \lfloor \frac{y_0}{c^i} \rfloor = \sum_{j=i}^n a_j c^{j-i}$ et $x = x_0 c^i$ sont des invariants directement validés.
 - On va maintenant prouver l'invariant $I(i, m) := m = x_0 \sum_{j=0}^{i-1} a_j c^j$.
 - * **Initialisation** avant la première itération, on a $I(0, 0, x_0, y_0)$ qui est vérifié car $0 = 0$.
 - * **Hérédité** si l'invariant est vérifié au début de la i ème itération on a $m + x(y \% c) = m + x_0 a_i c^i = m + x_0 \sum_{j=0}^i a_j c^j$ donc $I(i+1, m + x(y \% c))$ est vérifié.

Ainsi, en sortie de boucle, on a $m = x_0 \sum_{j=0}^n a_j c^j = x_0 y_0$.
3. En récursif, cela ne change pas la validité des preuves précédentes mais il faudrait présenter l'invariant différemment.

```
int multiplication(int x, int y, int c)
{
    if (y == 0) return 0;
    return x * (y % c) + multiplication(x * c, y / c, c);
}
```

■

Exercice 12.2 On considère un polynôme $P = \sum_{i=0}^n c_i X^i$ représenté comme un tableau de $n + 1$ coefficients tel que pour tout $i \in \llbracket 0, n \rrbracket$, $P[i] = c_i$.

On veut calculer $P(a)$ pour une valeur a . Pour simplifier, on va considérer ici que les valeurs sont toutes entières.

```

int horner(const int *P, int n, int a)
{
    int v = 0;
    for (int i = n; i >= 0; i--)
    {
        v = a * v + P[i];
    }
    return v;
}

```

1. Montrer que ce programme est correct.
2. Combien effectue-t-il de multiplications et d'additions ?
3. Comparer avec l'algorithme obtenu en ajoutant chaque $c_i a^i$ pour i croissant et en maintenant une variable pour a^i .

Démonstration.

1. On considère l'invariant $I(v, i) := v = c_{i+1} + c_{i+2}a + \dots + c_n a^{n-i-1}$
 - **Initialisation** avant la première itération on a $I(0, n)$ vérifié car $v = 0 = c_{n+1}$.
 - **Héritage** si l'invariant est vérifié en début d'itération, on passe à $I(av + c_i, i - 1)$ en fin d'itération or, si $v = c_{i+1} + \dots + c_n a^{n-i-1}$ on a bien $av + c_i = c_i + c_{i+1}a + \dots + c_n a^{n-i}$ et donc l'invariant est vérifié en fin d'itération.

En sortie de boucle, on a alors $I(v, -1)$ donc $v = c_0 + c_1a + \dots + c_n a^n = P(a)$.
2. On effectue $n + 1$ itérations et à chaque itération une multiplication et une addition, donc $n + 1$ multiplications et $n + 1$ additions.
3. Pour cet algorithme, on devrait faire une multiplication de plus à chaque itération pour maintenir a^i .

Exercice 12.3 On considère le problème du drapeau hollandais : étant donné un tableau t et un indice i , on note $p=t[i]$, on cherche à permuter les éléments de t de sorte qu'il y ait trois zones dans le tableau : les éléments $< p$, les éléments $= p$ puis les éléments $> p$.

- Ainsi, si on considère $t = [5, 2, 3, 5, 1, 4]$ et $p=t[2] = 3$ on pourra obtenir $[2, 1, 3, 5, 5, 4]$ à l'issue de cet algorithme.
1. Écrire un programme résolvant ce problème en temps linéaire.
 2. Prouver sa correction.

Démonstration.

Le programme :

```

void echange(int *t, int i, int j)
{
    int temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}

void drapeau(int *t, int nb, int i)
{
    int v = t[i];
    int l = 0;

```

```

int c = nb-1;
int r = nb-1;

while (l <= c)
{
    int w = t[l];
    if (w < v)
    {
        l = l+1;
    }
    else if (w == v)
    {
        echange(t, l, c);
        c = c-1;
    }
    else
    {
        echange(t, l, c);
        echange(t, c, r);
        c = c-1;
        r = r-1;
    }
}
}

```

L'idée de cet algorithme est d'avoir trois indices l , c et r (pour *left*, *center* et *right*) qui délimitent trois zones :

- celle des indices 0 à $l - 1$ qui contient des valeurs $< v$
- celle des indices $c + 1$ à r qui contient des valeurs égales à v
- celle des indices $r + 1$ à $nb - 1$ qui contient des valeurs $> v$.

Au départ, ces trois zones sont triviales et au fur à mesure de l'algorithme, elles augmentent jusqu'à couvrir l'ensemble du tableau. Comme soit l croît soit c décroît à chaque itération, on a la distance $c - l$ qui est un variant de boucle et on effectue exactement $|t|$ itérations.

On va montrer que le découpage des zones est effectivement un invariant :

- **Initialisation** au départ, comme on l'a vu, ces trois zones sont vides donc l'invariant est trivialement vérifié
- **Héritage** si l'invariant est vrai au début d'une itération, on va faire trois cas selon la position de w par rapport à v :

- ★ Si $w < v$ alors la zone des indices de 0 à l contient toujours des valeurs $< v$ et les autres zones n'ont pas bougé, l'invariant est encore vérifié.
- ★ Si $w = v$ alors on place cette valeur w au début de la zone centrale en faisant l'échange puis en décalant c vers la gauche. Les autres zones ne sont pas touchées, donc l'invariant est encore vérifié.
- ★ Si $w > v$ alors on veut le placer en tête de la zone de droite, mais l'élément qui s'y trouve est peut-être une valeur de la zone centrale si celle-ci est non triviale. On place alors cette valeur là en début de la zone centrale ce qui revient à la décaler d'un cran vers la gauche. La valeur w peut alors être échangée. La première zone n'a pas bougé et les autres zones vérifient encore les conditions voulues, l'invariant est donc vérifié.
!TODO(On devrait être plus précis ici)

On a donc validé l'invariant. En sortie de boucle, on a $x = y + 1 > y$ et donc les trois zones couvrent l'intégralité du tableau.

ALGORITHMS BY COMPLEXITY

MORE COMPLEX →

LETPAD QUICKSORT GIT MERGE SELF-DRIVING CAR GOOGLE SEARCH BACKEND

SPRAWLING EXCEL SPREADSHEET
BUILT UP OVER 20 YEARS BY A
CHURCH GROUP IN NEBRASKA TO
COORDINATE THEIR SCHEDULING

13. Complexité amortie

Source de l'image d'en-tête XKCD #1667

Introduction

Dans le cadre de l'étude des structures de données, il est fréquent de considérer non pas la complexité dans le pire des cas d'une opération mais celle d'une succession d'opérations divisée par le nombre d'opérations effectuées. Ainsi, on peut très bien avoir une opération ponctuellement plus coûteuse que les autres, mais en procédant ainsi on lisse le surcoût sur l'ensemble des opérations. On parle alors de **complexité amortie**.

■ **Remarque 13.1** Cette notion ne masque pas le fait qu'une opération puisse prendre ponctuellement plus de temps. Dans des contextes temps réel où il est important de maîtriser pleinement les complexités, il est peu judicieux d'utiliser de telles complexités. Par exemple, dans une visualisation en 3D, pour maintenir un débit constant d'images par secondes, chaque image doit prendre un temps similaire. Se reposer sur une structure de donnée ayant une faible complexité amortie mais une complexité dans le pire des cas importante, c'est risquer d'avoir des saccades avec une image qui prendrait plus de temps pour être calculée.

On va reprendre ici deux exemples simples déjà traité par ailleurs.

1.1 Implémentation d'une file avec deux piles

On considère une file réalisée avec deux piles `in` et `out` (voir le chapitre Piles et files).

Comme on l'a vu, on enfile en $O(1)$ en empilant sur la pile `in` et on défile :

- soit en $O(1)$ en défilant sur `out` si elle est non vide
- soit en $O(n)$ en déversant `in` dans `out` puis en se ramenant au cas précédent.

Ainsi, en pire cas, on est en $O(n)$ pour l'opération *défiler* mais la bascule permettra de faire ensuite cette même opération en $O(1)$.

1.2 Les tableaux dynamiques

Les tableaux dynamiques sont une structure de données de haut niveau pour implémenter le type abstrait des listes. La différence principale entre cette structure de données et celle des tableaux de C est qu'on peut ajouter et supprimer des éléments.

Un tableau dynamique d'entiers est un triplet (t, c, l) où t est un tableau de taille c , appelée la capacité du tableau dynamique, et l est un autre entier représentant la longueur logique du tableau. A tout moment $c \geq l$. Dans t il y a ainsi $c - l$ cases déjà allouées qui permettent de rajouter un élément en temps constant. Quand $c = l$, on alloue une nouvelle zone mémoire, souvent de taille $2c$, on déplace le tableau t dans cette zone et on a donc pour cet ajout prévu un certain nombre de cases d'avance.

La figure suivante présente une succession d'ajouts :

$c = 0$	$l = 0$									
$c = 1$	$l = 1$	<table border="1"><tr><td>1</td></tr></table>	1							
1										
$c = 2$	$l = 2$	<table border="1"><tr><td>1</td><td>2</td></tr></table>	1	2						
1	2									
$c = 4$	$l = 3$	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td></td></tr></table>	1	2	3					
1	2	3								
$c = 4$	$l = 4$	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	1	2	3	4				
1	2	3	4							
$c = 8$	$l = 5$	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td></td><td></td><td></td></tr></table>	1	2	3	4	5			
1	2	3	4	5						

■ **Remarque 13.2** Une implémentation de ces opérations est proposée dans le programme suivant :


```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *t;
    size_t c;
    size_t l;
} tableau_dynamique ;

tableau_dynamique tableau_dynamique_creer()
{
    tableau_dynamique d;

    d.t = NULL;
    d.c = 0;
    d.l = 0;

    return d;
}

void tableau_dynamique_ajout(tableau_dynamique *d, int x)
{
    if (d->c == d->l) {
        if (d->c == 0)
        {
            d->c = 1;
            d->t = malloc(sizeof(int));
        }
        else
        {
            d->c = 2 * d->c;
            d->t = realloc(d->t, d->c * sizeof(int));
        }
    }
    d->t[d->l] = x;
    d->l++;
}

void tableau_dynamique_print(tableau_dynamique d)
{
    printf("c=%d\ tl=%d\ t|", d.c, d.l);
    for (size_t i = 0; i < d.l; i++) {
        printf("%d|", d.t[i]);
    }
    for (size_t i = d.l; i < d.c; i++) {
        printf(" |");
    }
    printf("\n");
}

int main(void) {
    tableau_dynamique d = tableau_dynamique_creer();
    tableau_dynamique_print(d);
    for (int i = 1; i < 6; i++) {
        tableau_dynamique_ajout(&d,i);
        tableau_dynamique_print(d);
    }
}
```

Ce programme, une fois exécuté produit la sortie suivante qui permet de retrouver exactement le comportement attendu :

```
c=0 l=0 |
c=1 l=1 |1|
c=2 l=2 |1|2|
c=4 l=3 |1|2|3|
c=4 l=4 |1|2|3|4|
c=8 l=5 |1|2|3|4|5| | | |
```

■

Avec une analyse en pire cas, l'ajout d'un élément peut entraîner un doublement de taille et une recopie, donc coûter $O(c)$ opérations où c est la capacité avant doublement. Cependant, on constate que ce doublement permettra plusieurs ajouts ensuite en temps constant, ainsi, le pire cas ne semble pas le plus pertinent.

II Techniques de calcul

II.1 Cadre

On considère une structure de données, pour simplifier initialement vide, et une successions o_1, \dots, o_n d'opérations **valides** sur cette structure de donnée. En reprenant l'exemple précédent, cela peut être une succession d'ajouts ou de suppressions dans une file mais sans jamais essayer de retirer des éléments à une file vide.

On considère de plus que pour chaque opération o_i , on connaît le coût temporel c_i en terme d'opérations élémentaires. La question que l'on se pose est d'estimer $\frac{1}{n} \sum_{i=1}^n c_i$, c'est-à-dire le coût moyen de chaque opération. L'objectif est de montrer que ce coût moyen est $O(f(n))$, avec assez souvent $O(1)$, c'est-à-dire que $\sum_{i=1}^n c_i = O(nf(n))$.

II.2 Calcul naïf

Une approche naïve consiste à calculer directement la somme des coûts. Tout le problème ici est qu'on ne connaît pas les opérations et l'ordre dans lequel elles ont été effectuées.

II.2.i Exemple de la file avec deux piles

On va commencer par calculer c_i pour les différentes opérations :

- S'il s'agit d'un ajout sur **in** ou d'un dépilement sur **out**, on a $c_i = 1$
- Si l'opération est un dépilement nécessitant un renversement, on a $c_i = 2p + 1$ où p est la taille de **in**.

Le problème du calcul naïf ici est de connaître l'ordre des opérations. On peut regrouper les opérations o_1, \dots, o_n par paquet : ajouts, suppression avec renversement, ajouts ou suppressions sans renversement, suppression avec renversement... On va noter p_i la taille de **in** avant le i ème renversement. On a donc p_{i+1} ajouts entre le i ème et le $(i+1)$ ème renversement. Ainsi, on a effectué $p_i - 1$ suppressions sans renversements entre les deux.

Comme cela correspond au pire cas, on peut supposer que o_n est une suppression avec renversement.

On obtient alors

$$\begin{aligned} \sum_{i=1}^n c_i &= p_1 + (2p_1 + 1) + p_2 + (p_1 - 1) + (2p_2 + 1) \\ &\quad + \cdots + p_r + (p_{r-1} - 1) + (2p_r + 1) \\ &\leq 4(p_1 + p_2 + \cdots + p_r) \end{aligned}$$

Pour conclure, il est nécessaire de relier le nombre d'opérations aux p_i . Or, on a

$$\begin{aligned} n &= p_1 + 1 + p_2 + p_1 - 1 + \cdots + p_r + p_{r-1} - 1 + 1 \\ &= 2p_1 + 2p_2 + \cdots + 2p_{r-1} + p_r \geq p_1 + \cdots + p_r \end{aligned}$$

Ainsi $\frac{1}{n} \sum_{i=1}^n c_i \leq 4$.

■ **Remarque 13.3** Cette analyse est assez pénible à mettre en oeuvre car elle demande de compter avec précision ce qui se passe.

II.2.ii Exemple du tableau dynamique

Ici, les opérations sont toutes des ajouts d'éléments, mais certains provoquent une recopie tandis que d'autres sont en temps constant. Cependant, on a la même distribution des coûts, la suite des c_i est toujours $1, 2, 3, 1, 5, 1, 1, 1, 9, 1, 1, 1, \dots$

Plus précisément, $c_i = \begin{cases} 2^p + 1 & \text{si } i = 2^p + 1 \\ 1 & \text{sinon} \end{cases}$.

Donc si $2^{p-1} < n \leq 2^p$ on a $\sum_{i=1}^n c_i = n + \sum_{k=1}^{p-1} 2^k = n + 2^p \leq 3n$.

On en déduit que $\frac{1}{n} \sum_{i=1}^n c_i \leq 3$ c'est-à-dire que c'est un $O(1)$.

II.2.iii Améliorer la méthode naïve

Une amélioration de la méthode naïve va consister à considérer non pas le coût réel c_i de l'opération o_i mais un coût amorti c'_i tel qu'à la fin des opérations on ait $\sum_{i=1}^n c_i \leq \sum_{i=1}^n c'_i$ où cette dernière somme est facile à majorer.

On fera souvent en sorte que la majoration soit valide tout au long des opérations, c'est-à-dire que

$$\forall k \in \llbracket 1, n \rrbracket, \sum_{i=1}^k c_i \leq \sum_{i=1}^k c'_i$$

II.3 Méthode du banquier

On va imaginer maintenant que les opérations coûtent un certain nombre de crédits, comme si l'ordinateur marchait avec des pièces de monnaie.

Des opérations vont alors accumuler des crédits dans une réserve et d'autres opérations vont consommer ces crédits. L'idée est de payer un peu plus pour chaque opération peu coûteuse afin de couvrir les coûts des opérations coûteuses.

Précisément, on considère pour chaque opération o_i :

- une quantité a_i de crédits qui seront accumulés dans la réserve
- et une quantité d_i de crédits qu'il faudra dépenser.

On pose alors $c'_i = c_i + a_i - d_i$. On s'arrange pour faire en sorte que $c'_i \leq Af(n)$ où A est une constante. Dans le cas d'une complexité constante amortie, on s'assurera que $c'_i \leq A$.

On va maintenir un invariant sur les crédits qui est que

$$\forall k \in \llbracket 1, n \rrbracket, \sum_{i=1}^k a_i \geq \sum_{i=1}^k d_i$$

Cela signifie qu'à tout moment la réserve de crédit est positive ou nulle : on ne dépense pas de crédits n'ayant pas été accumulés.

On a alors directement la majoration voulu de $\sum_i c_i$: soit $k \in \llbracket 1, n \rrbracket$,

$$\sum_{i=1}^k c'_i = \sum_{i=1}^k c_i + \left(\sum_{i=1}^k a_i - \sum_{i=1}^k d_i \right) \geq \sum_{i=1}^k c_i$$

II.3.i Exemple de la file avec deux piles

Pour chaque élément enfilé on va allouer deux crédits correspondant au futur dépilement de `in` puis empilement sur `out` qui auront lieu lors du renversement.

On pose donc pour chaque opération o_i , $c'_i = c_i + a_i - d_i$ où :

- $a_i = 2$ et $d_i = 0$ si l'opération est un ajout
- $a_i = d_i = 0$ si l'opération est un défilement ne nécessitant pas de renversement
- $a_i = 0$ et $d_i = 2p$ où p est la taille de `in` dans le cas d'un défilement provoquant un renversement.

On a donc $\sum_{i=1}^k a_i = 2A$ où A est le nombre d'ajouts effectués par les k premières opérations. Si on a effectué des renversements successifs de p_1, p_2, \dots, p_r éléments dans `in`, on a

$$\begin{aligned} \sum_{i=1}^k d_i &= 2p_1 + \cdots + 2p_r \\ &= 2(p_1 + \cdots + p_r) \\ &\leq 2A \\ &\leq \sum_{i=1}^k a_i \end{aligned}$$

La majoration venant du fait qu'un élément n'est ajouté qu'une seule fois dans `in` et donc que la somme $p_1 + \cdots + p_r$ des longueurs de `in` au cours des opérations ne peut dépasser le nombre A d'éléments empilés.

On a ainsi vérifié l'invariant sur les crédits et on a donc la majoration $\sum_{i=1}^k c_i \leq \sum_{i=1}^k c'_i$.

Pour conclure, il suffit de calculer c'_i selon les cas :

- Si l'opération est un ajout, on a $c'_i = 1 + 2 - 0 = 3$.
- Si l'opération est une suppression sans renversement, on a $c'_i = 1$
- Si l'opération est une suppression avec renversement, on a $c'_i = 2p + 1 + 0 - 2p = 1$

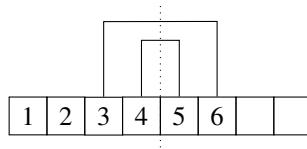
Dans tous les cas, on a donc $c'_i \leq 3$.

On en déduit la majoration $\sum_{i=1}^n c_i \leq \sum_{i=1}^n c'_i \leq 3n$ qui est plus fine que la majoration obtenue précédemment.

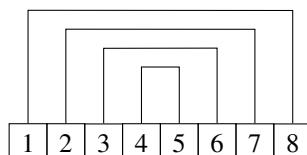
II.3.ii Exemple du tableau dynamique

On va considérer qu'on alloue 2 crédits pour chaque opération d'ajout d'un élément :

- Un crédit pour la première recopie de cet élément
- Un crédit pour copier l'élément qui lui est symétrique par rapport à la demi-capacité :



Ainsi, juste avant un doublement de capacité, on dispose d'un crédit pour chaque recopie :



Plus précisément, on définit pour chaque opération $o_i : c'_i = c_i + a_i - d_i$ où $a_i = \begin{cases} 1 & \text{si } i = 1 \\ 2 & \text{sinon} \end{cases}$
et $d_i = \begin{cases} 2^p & \text{si } i = 2^p + 1 \\ 0 & \text{sinon} \end{cases}$.
On a donc $c'_i = 3$ dans tous les cas.

Théorème II.1 $\forall n \in \mathbb{N}^*, \sum_{i=1}^n a_i \geq \sum_{i=1}^n d_i$.

Démonstration.

On a $\sum_{i=1}^n a_i = 2n - 1$ et si $2^p < n \leq 2^{p+1}$,

$$\begin{aligned} \sum_{i=1}^n d_i &= d_2 + d_3 + d_5 + \cdots + d_{2^p+1} \\ &= 1 + 2 + 2^2 + \cdots + 2^p \\ &= 2^{p+1} - 1 \leq 2n - 1 \end{aligned}$$

L'inégalité est bien vérifiée. ■

Ainsi,

$$3n = \sum_{i=1}^n c'_i = \sum_{i=1}^n c_i + \left(\sum_{i=1}^n a_i - \sum_{i=1}^n d_i \right) \geq \sum_{i=1}^n c_i$$

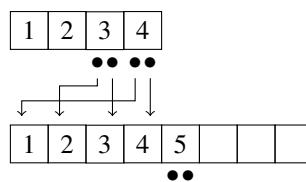
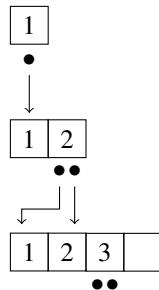
On retombe sur une complexité amortie en $O(1)$.

II.3.iii Présentation de la méthode

La présentation qui vient d'être faite de cette méthode est assez lourde bien que précise. En général, on se contente de présenter informellement la distribution de crédits en s'assurant qu'on a couvert les opérations coûteuses.

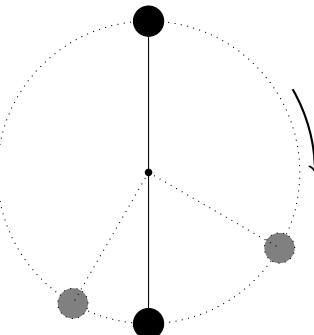
Une manière de bien s'en assurer est de *décorer* les structures de données en rendant les crédits apparents.

Si on reprend les tableaux dynamiques, on va rajouter sous une case le nombre de crédits dont elle dispose. Si on reprend la séquence d'ajouts précédente on pourra la représenter ainsi :



II.4 Méthode du potentiel

L'idée de la méthode du potentiel est de définir une fonction sur Φ qui associe à chaque valeur de la structure de données un nombre réel. Ce réel correspond à la notion physique de potentiel : si on considère un pendule simple dans sa trajectoire son énergie potentielle est maximale au sommet quand sa vitesse, et donc son énergie cinétique est nulle. À l'inverse son énergie potentielle est nulle au point le plus bas et sa vitesse maximale.



L'idée ici est que le potentiel va correspondre aux gains de temps obtenus par des opérations peu coûteuses. Chaque opération va ainsi faire évoluer le potentiel. On peut prendre l'image d'un empilement de tuiles de bois, comme des Kapla, chaque ajout d'une tuile de bois augmente le potentiel jusqu'à un éventuel effondrement. On retrouve ici la dualité entre les opérations rapides et les opérations coûteuses liées à un basculement.

Précisement, on va noter t_i la structure obtenue à l'issue de l'opération o_i . On considère que t_0 est la structure vide et $\Phi(t_0) = 0$. On pose $c'_i = c_i + \Phi(t_i) - \Phi(t_{i-1})$ ce qui correspond à considérer la différence de potentiel.

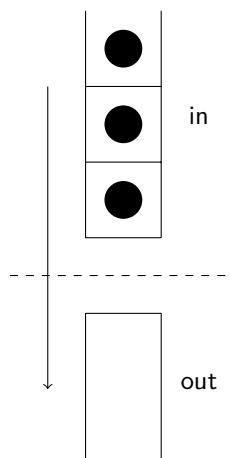
On a donc

$$\sum_{i=1}^k c'_i = \sum_{i=1}^k c_i + \sum_{i=1}^k \Phi(t_i) - \Phi(t_{i-1}) = \sum_{i=1}^k c_i + \Phi(t_k)$$

Pour conclure par une majoration de $\sum_{i=1}^k c_i$, il suffit de s'assurer que le potentiel ne soit jamais négatif.

II.4.i Exemple de la file avec deux piles

On va poser $\Phi(t) = 2|\text{in}|$ où $|\text{in}|$ est la taille de la pile d'entrée. Pour reprendre le parallèle physique, on peut s'imaginer que la pile **in** est au dessus de la pile **out** et que les éléments menacent de s'y déverser. Ce qui permet d'avoir l'image du potentiel comme pour le pendule au sommet de sa trajectoire.



On a directement la majoration car $\Phi(t) \geq 0$.

Pour conclure, il faut calculer c'_i dans chaque cas :

- Si l'opération est un ajout, in augmente d'un élément et donc $c'_i = c_i + 2 = 3$
- Si l'opération est une suppression sans renversement, le potentiel ne change pas et $c'_i = 1$
- Si l'opération est une suppression avec renversement et que in contient p éléments, on a $c'_i = 2p + 1 + 0 - 2p = 1$.

On retombe exactement sur le calcul précédent avec la méthode du banquier.

II.4.ii Exemple du tableau dynamique

On pose ici $\Phi(t) = 2l(t) - c(t)$ où $l(t)$ est la longueur de t et $c(t)$ est sa capacité.

Ici, ce potentiel se comprend mieux en écrivant $2(l(t) - c(t)/2)$ car $l(t) - c(t)/2$ correspond aux éléments ajoutés après doublement.

Comme $c(t) \geq l(t) \geq \frac{c(t)}{2}$ on a $\Phi(t) \geq 0$ ce qui valide la majoration.

On calcule c'_i dans les deux cas d'ajout :

- Si c'est un ajout sans doublement, alors $l(t_i) = 1 + l(t_{i-1})$, $c(t_i) = c(t_{i-1})$ et ainsi

$$\begin{aligned} c'_i &= c_i + \Phi(t_i) - \Phi(t_{i-1}) \\ &= 1 + 2l(t_{i-1}) - c(t_{i-1}) - 2l(t_{i-1}) + c(t_{i-1}) = 1 \end{aligned}$$

- Si c'est un ajout avec doublement de capacité, alors $l(t_{i-1}) = c(t_{i-1}) = A$, $l(t_i) = 1 + l(t_{i-1}) = 1 + A$, $c(t_i) = 2c(t_{i-1}) = 2A$ et $c_i = A + 1$. On a donc :

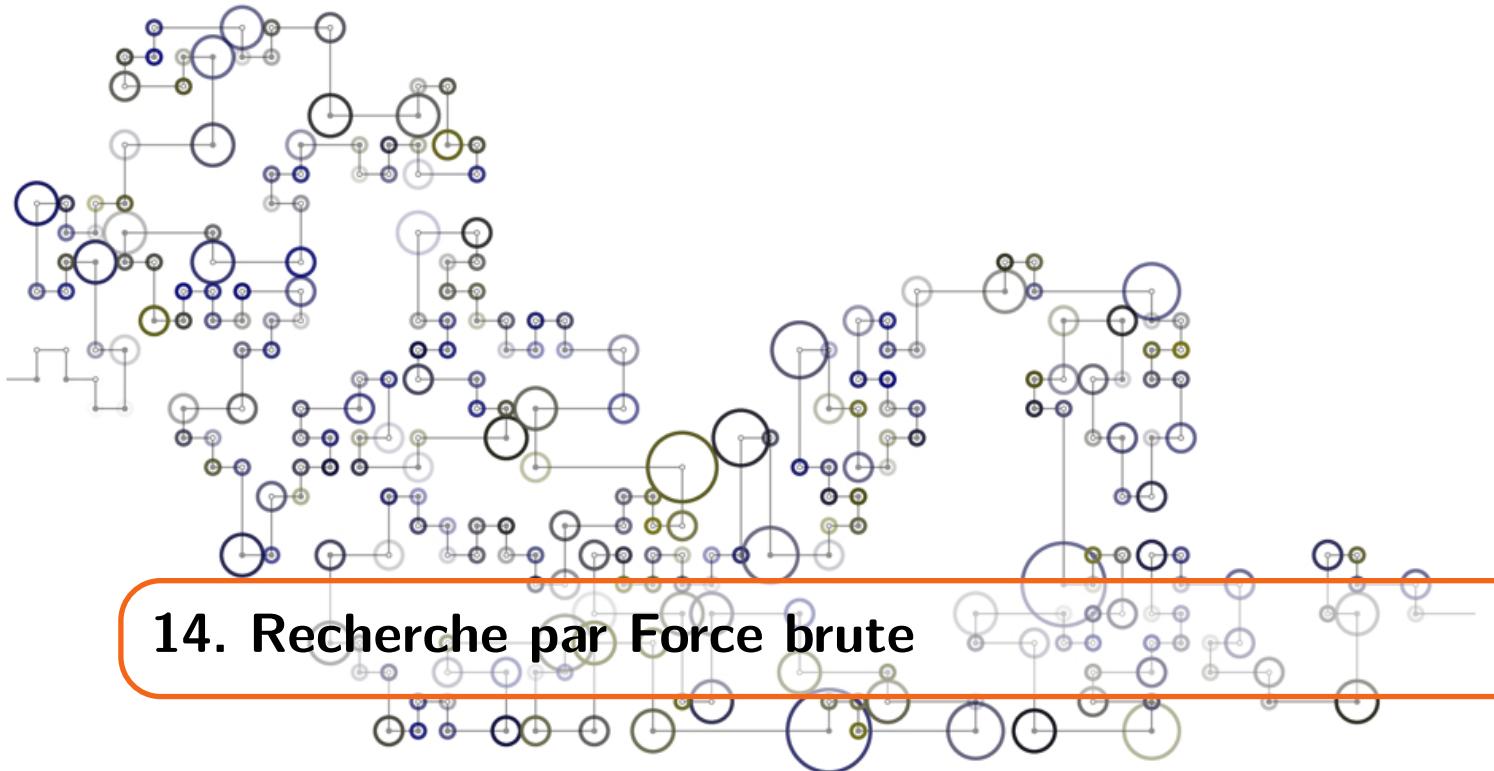
$$\begin{aligned} c'_i &= c_i + \Phi(t_i) - \Phi(t_{i-1}) \\ &= A + 1 + 2(A + 1) - 2A - 2A + A \\ &= 3 \end{aligned}$$

Ainsi, $3n \geq \sum_{i=1}^n c'_i \geq \sum_{i=1}^n c_i$. On retombe sur la majoration voulue.

II.4.iii Comparatif des méthodes

On remarque que la méthode du potentiel et la méthode du banquier sont très proches. En fait, on peut montrer qu'elles sont équivalentes. La différence entre les deux est que la méthode du banquier demande de regarder avec précision les opérations pour lesquelles il va falloir louer des crédits alors que pour la méthode du potentiel il suffit de trouver son expression. Cependant, trouver l'expression du potentiel est parfois plus complexe que de définir les crédits.

On cherchera donc la méthode la mieux adaptée aux problèmes considérés. Dans l'exemple de la file, le potentiel est très facilement défini, dans le cas du tableau dynamique, la méthode du banquier peut paraître plus naturelle.



14. Recherche par Force brute

Source image : <https://www.flickr.com/photos/x6e38/3440634940/>

I Principe

I.1 Problème de décision et exploration exhaustive

Considérons un problème du type trouver un $x \in V$ vérifiant une propriété $P(x)$.

- **Exemple 14.1**
 - V est l'ensemble des chaînes de caractères et P vérifié si si une chaîne est un mot de passe qu'on cherche.
 - V est l'ensemble des indices possibles dans un tableau t et P vérifie si la valeur à l'indice i est une valeur x que l'on cherche.
 - V est l'ensemble des grilles complétées d'un problème de Sudoku et P vérifie si la grille est valide.
 - V est l'ensemble d'assemblages de pièces d'un puzzle et P vérifie si le puzzle est correcte, c'est-à-dire si deux pièces côtés à côtés ont des côtés compatibles.

Dans certains problèmes, un tel x n'est pas unique et on cherche à tous les énumérer.

Une recherche par force brute ou recherche exhaustive, consiste à énumérer l'ensemble V jusqu'à obtenir une solution en testant P pour chaque valeur rencontrée.

Des problèmes précédents, la recherche linéaire est le plus simple et le programme suivant est caractéristique d'une recherche exhaustive :

```
exception Trouve of int

let recherche t x =
  try
    for i = 0 to Array.length t - 1 do
      if t.(i) = x
      then raise (Trouve i)
```

Ocaml

```

done;
raise Not_found
with Trouve i -> i

```

La forme usuelle sera alors

```

pour chaque v dans V
  si P(v) est vérifié
    s'arrêter avec la solution v

```

On rappelle que pour pouvoir s'arrêter au cours de l'énumération en OCaml, si on programme en impératif, on utilise en général des exceptions comme dans le chapitre Exceptions en OCaml.

Cela pose naturellement la question de l'énumération des éléments de V . Si c'est immédiat dans l'exemple peu pertinent de la recherche dans un tableau, c'est beaucoup plus complexe pour l'énumération des assemblages de pièces d'un puzzle, par exemple.

Pour la recherche du mot de passe, on pourrait commencer par énumérer les chaînes de longueur 1, puis de longueur 2, et ainsi de suite.

Le plus souvent, l'ensemble V est fini (pour les mots de passe, cela peut consister à limiter la longueur maximale du mot de passe). Ainsi, une recherche par force brute effectue $O(|V|)$ itérations.

I.2 Problème d'optimisation et exploration exhaustive

On retrouve la notion d'exploration exhaustive ou force brute pour des problèmes d'optimisation. Il s'agit de problèmes de la forme : déterminer $x \in V$ tel que $f(x)$ soit minimale ou maximale.

L'exploration exhaustive consiste alors à calculer toutes les images par f des éléments de V afin de déterminer un extremum.

II Recherche par retour sur trace (backtracking)

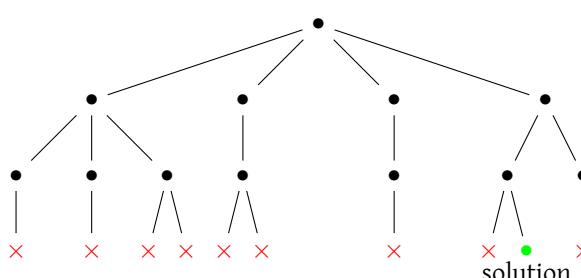
II.1 Construction itérative de candidats

Dans de nombreux cas, l'ensemble V peut se décrire par un processus itératif de construction de ses éléments. Une manière de voir cela est de parler de positions et de mouvements, ou coups.

Par exemple, considérons un puzzle comme le puzzle Eternity II constitué de 256 pièces carrées. Une configuration finale du puzzle consiste à avoir placé les 256 pièces. Parmi celles-ci, les configurations valides sont celles satisfaisant les contraintes de chaque côté.

Comme pour tous les puzzles, la position initiale est un plateau vide et chaque mouvement consiste à placer une pièce disponible dans un emplacement disponible. Cela correspond à la manière dont on procéderait à la main :

Ainsi, on peut représenter la construction de V sous la forme d'un arbre dont les nœuds sont les positions et les arêtes les mouvements. Les positions complètes sont les feuilles de l'arbre, elles correspondent aux éléments de V et ce sont donc celles-ci qu'on va explorer pour y trouver une solution.



L'avantage de cette représentation arborescente est qu'elle découle naturellement d'un parcours récursif des positions.

Tout d'abord, il va falloir définir un type de positions partielles comme on a pu le voir dans le chapitre Options en OCaml.

Comme indiqué dans ce chapitre, si le type d'une grille de Sudoku remplie est `int array array`, pour pouvoir représenter des grilles en cours de remplissage, on va utiliser le type `int option array array`. On rappelle que l'utilisation de la valeur `None` permettra de représenter une partie non construite comme une case vide.

On considère ainsi un type `position` des positions partielles, un type `mouvement` et des fonctions

```
(* les mouvements accessibles depuis une position.
[] si la position est complète *)
val mouvements : position -> mouvement list
(* applique un mouvement *)
val applique : position -> mouvement -> position
(* vérifie si une position est complète *)
val complete : position -> bool
(* vérifie si une position complète est valide *)
val valide : position -> bool
```

OCaml

Schématiquement, un algorithme d'énumération aura la structure suivante en OCaml :

```
exception Solution of position

let rec enumere pos =
  if complete pos
  then begin
    if valide pos
    then raise (Solution pos)
  end else
    List.iter (fun mouv -> enumere (applique pos mouv))
              (mouvements pos)
```

OCaml

■ **Remarque 14.1** Ici, l'usage de `List.iter` permet de faire l'équivalent d'une boucle `for` mais sur une liste. Pour s'en passer, on peut écrire une fonction auxiliaire récursive ou, quand les mouvements sont en petit nombre, une conversion en `array` suivie d'une boucle `for`.

Si la position initiale est `pos0`, on pourra résoudre le problème ainsi :

```
let resout () =
  try
    enumere pos0;
    raise Not_found
  with Solution pos -> pos
```

OCaml

Cet algorithme de parcours des solutions est appelé le *retour sur trace*, ou *backtracking* en anglais. Il tire partie de la récursivité pour remonter les positions après avoir essayé en vain une construction.

■ **Remarque 14.2** Il est possible de reprendre la construction précédente avec des données mutables et une manière de *défaire* les mouvements :

```

Ocaml

val applique : position -> mouvement -> unit
val defaire : position -> mouvement -> unit

let rec enumere pos =
  if complete pos
  then begin
    if valide pos
    then raise (Solution pos)
  end else
    List.iter (fun mouv ->
      applique pos mouv;
      enumere pos;
      defaire pos mouv)
    (mouvements pos)
```

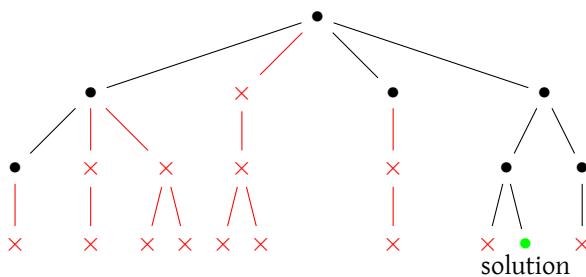
II.2 Évaluation partielle et raccourcis

Si on reprend la construction itérative précédente, on se rend compte qu'elle n'est pas très intelligente : faut-il remplir l'intégralité d'un puzzle avant de se rendre compte qu'il est invalide en raison des deux premières pièces ?

On peut donc raffiner l'approche précédente en introduisant une notion de mouvements valides qui sont les mouvements qui préservent la correction partielle.

En pratique, il suffit de remplacer la fonction `mouvements` par une fonction `mouvements_valides`.

C'est un changement réduit qui peut avoir un grand impact sur l'arbre des positions. En reprenant l'illustration précédente, on peut imaginer que cela reviendrait à ne pas parcourir les mouvements rouges et les sous-arbres associés :



II.2.i Problème : résolution de Sudoku

La recherche par retour sur trace se prête très bien à la résolution de problèmes comme le Sudoku. On va ici tout simplement tenter de remplir chaque case du haut vers le bas tant qu'on satisfait les contraintes du Sudoku. Le programme sera ainsi très proche de la résolution des huit reines.

Commençons par rappeler le principe du Sudoku :

- On part d'une grille de 81 cases réparties en une grille de 3x3 sous-grilles de 3x3 cases et comportant des chiffres de 1 à 9 dans certaines cases.

1								6
		6		2		7		
7	8	9	4	5		1		3
			8		7			4
				3				
	9				4	2		1
3	1	2	9	7			4	
	4			1	2		7	8
9		8						

- L'objectif est de remplir chaque case avec un chiffre de 1 à 9 de sorte que chaque ligne, chaque colonne et chaque sous-grille 3x3 comporte une et une seule fois chaque chiffre.
- Un sudoku admet une unique solution.

Pour représenter une grille de Sudoku en OCaml on utilise un `(int option) array array`, la valeur `None` signifiant que la case est vide et la valeur `Some x` qu'elle est remplie avec la valeur x .

OCaml

```
type grille = (int option) array array
```

On fait le choix de représenter la grille par un tableau de lignes, ce qui signifie que pour accéder à la case de coordonnée (x, y) dans g il faut écrire $g.(y).(x)$.

Le problème donné précédemment est alors représenté par la valeur suivante :

OCaml

```
let probleme = [
  [ Some 1; None; None; None; None; None; None; None; Some 6 ];
  [ None; None; Some 6; None; Some 2; None; Some 7; None; None ];
  [ Some 7; Some 8; Some 9; Some 4; Some 5; None; Some 1; None; Some 3 ];

  [ None; None; None; Some 8; None; Some 7; None; None; Some 4 ];
  [ None; None; None; None; Some 3; None; None; None; None ];
  [ None; Some 9; None; None; None; Some 4; Some 2; None; Some 1 ];

  [ Some 3; Some 1; Some 2; Some 9; Some 7; None; None; Some 4; None ];
  [ None; Some 4; None; None; Some 1; Some 2; None; Some 7; Some 8 ];
  [ Some 9; None; Some 8; None; None; None; None; None; None ];
]
```

Afin de définir la fonction de résolution, on définit une première fonction suivant de signature :

OCaml

```
val suivant : grille -> (int * int) -> (int * int) option
```

telle que l'appel à `suivant g (x, y)` renvoie `Some (xi, yi)` quand (x_i, y_i) sont les coordonnées de la prochaine case libre, dans l'ordre gauche à droite puis haut vers bas, après (x, y) ou `None` quand il n'existe pas de telle case libre. Cela signifie alors que la grille est entièrement remplie.

```
let rec suivant g (x,y) =
  if y > 8
  then None
  else if g.(y).(x) = None
  then Some (x,y)
  else if x < 8 then suivant g (x+1, y)
  else suivant g (0, y+1)
```

On définit également une fonction valide de signature

```
val valide : grille -> int -> int -> bool
```

telle que l'appel à `valide g x y` renvoie `true` si et seulement si la valeur placée en coordonnée (x, y) n'invalidise pas la grille. Ne pas prendre cette valeur en paramètre permettant d'écrire un peu plus simplement cette fonction. La fonction est assez directe, étant donné (x, y) on va parcourir sa ligne, sa colonne et sa sous-grille pour vérifier qu'un nombre n'a pas été placé deux fois à l'aide d'un tableau de drapeaux :

```
let valide g x y =
  let v = ref true in
  let vus_colonne = Array.make 9 false in
  for y0 = 0 to 8 do
    match g.(y0).(x) with
    | None -> ()
    | Some k ->
        if vus_colonne.(k-1)
        then v := false;
        vus_colonne.(k-1) <- true
  done;
  let vus_ligne = Array.make 9 false in
  for x0 = 0 to 8 do
    match g.(y).(x0) with
    | None -> ()
    | Some k ->
        if vus_ligne.(k-1)
        then v := false;
        vus_ligne.(k-1) <- true
  done;
  let vus_grille = Array.make 9 false in
  let xb = (x / 3) * 3 in
  let yb = (y / 3) * 3 in
  for xd = 0 to 2 do
    for yd = 0 to 2 do
      match g.(yb+yd).(xb+xd) with
      | None -> ()
      | Some k ->
          if vus_grille.(k-1)
          then v := false;
          vus_grille.(k-1) <- true
    done
  done;
!v
```

On peut alors définir la fonction `resout` qui va résoudre le Sudoku en effectuant tous les remplissages tant qu'on a une grille valide. Dès qu'une solution est trouvée, on s'arrête. Pour cela, on utilise le mécanisme des exceptions pour permettre une sortie prématuée. On a fait le choix de travailler en place dans la grille, ainsi à la fin de l'exécution de la fonction, la grille correspond à la solution.

```
exception Solution

let resout g =
  let rec aux xi yi = match suivant g (xi, yi) with
    | None -> raise Solution
    | Some (x,y) ->
        for i = 1 to 9 do
          g.(y).(x) <- Some i;
          if valide g x y
          then begin
            aux x y
          end
        done;
        g.(y).(x) <- None
  in
  try
    aux 0 0
  with Solution -> ()
```

OCaml

II.3 Énumération de toutes les solutions

Le problème précédent du Sudoku n'avait par définition qu'une unique solution. Cependant, il existe des problèmes pour lesquels plusieurs solutions existent et pour lesquels on souhaite les énumérer.

La fonction précédente pourra alors devenir :

```
let rec enumere pos =
  if complete pos
  then begin
    if valide pos
    then [ pos ]
    else []
  end
  else
    List.concat
      (List.map (fun mouv -> enumere (applique pos mouv))
        (mouvements_valides pos))
```

OCaml

■ **Remarque 14.3** La fonction `List.concat` permet de concaténer une 'a list list d'un coup comme dans :

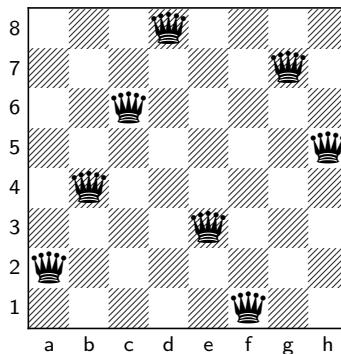
```
OCaml # List.concat [ [1;2]; [3]; []; [4;5;6] ];;
- : int list = [1; 2; 3; 4; 5; 6]
```



II.3.i Problèmes des huit reines

L'exemple classique de ce problème est celui des huit reines : étant donné un échiquier, peut-on placer huit reines de sorte qu'aucune reine ne puisse prendre une autre reine ? Plus précisément : sur un plateau de 8x8 cases, peut-on placer huit pions tels que deux pions quelconques ne soient jamais sur la même ligne ou la même diagonale ?

Exemple de solution :



Ce problème admet effectivement des solutions partielles en ne considérant que k reines à placer. Pour énumérer les solutions, on peut même se contenter de solutions partielles où les k reines sont placées sur les k premières rangées.

Voici ainsi un algorithme pour énumérer les solutions :

- Supposons que k reines aient été placées et qu'on dispose d'une solution partielle.
 - * Si $k = 8$ alors toutes les reines sont placées et la solution est complète, on la compléte
 - * Sinon, on continue la recherche pour chaque position de la $k + 1$ reine sur la $k + 1$ rangée qui préserve le fait d'être une solution partielle.

Ici, quand on dit qu'on continue la recherche, ce qu'on signifie, c'est qu'on effectue un appel récursif.

Pour programmer cette méthode, on va définir une fonction récursive de signature :

```
OCaml | val resout_reines : (int * int) list -> (int * int) list list
```

Un appel à `resout_reines part` va ainsi renvoyer la liste des solutions complètes construites à partir de la solution partielle `part`. Les solutions sont représentées par des listes de couples de coordonnées sur l'échiquier, donc dans $[[0; 7]]^2$

Voici une implémentation où on explore les solutions à l'aide d'une boucle impérative dans l'appel récursif. La fonction `valide` permet de tester si le placement d'une reine est possible avant d'effectuer un appel.

```
OCaml | let rec valide (x1,y1) l =
  match l with
  | [] -> true
  | (x2,y2)::q ->
    x1 <> x2 && abs (x2-x1) <> abs(y2-y1) && valide (x1,y1) q

  let rec resout_reines part =
    let k = List.length part in
```

```

if k = 8
then [ part ]
else begin
  let resultats = ref [] in
  for x = 0 to 7 do
    let essai = (x,k) :: part in
    if valide (x,k) part
    then begin
      resultats := (resout_reines essai) @ !resultats;
    end
  done;
  !resultats
end

```

et, ici, une autre implémentation purement récursive à l'aide d'une fonction récursive.

```

let rec resout_reines part =
  let k = List.length part in
  if k = 8
  then [ part ]
  else
    let rec aux x acc =
      if x < 0
      then acc
      else let essai = (x,k) :: part in
            let nacc = if valide (x,k) part
                       then (resout_reines essai) @ acc
                       else acc in
            aux (x-1) nacc
    in
    aux 7 []

```

OCaml

Une partie de l'arbre de recherche est présenté sur l'image suivante :

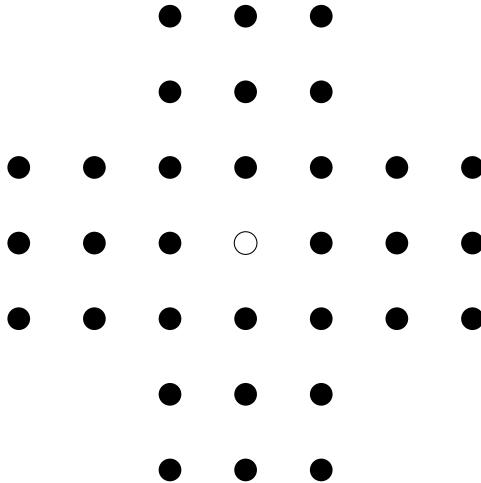
Arbre de recherche pour les huit reines

L'arbre complet comporte 2057 noeuds dont 92 feuilles correspondant aux solutions du problème. À titre de comparaison, l'arbre exhaustif correspondant à faire tous les choix de placement à raison d'une reine par ligne compterait $8^8 = 16777216$ noeuds. On voit bien que le backtracking est plus économique en exploration.

II.4 TP : tours du cavalier

II.5 TP : jeu du solitaire

On considère ici le jeu du solitaire. On a un plateau comportant 33 emplacements et initialement 32 pions, représenté par des ronds blancs, et un emplacement vide au centre :



Les différents mouvements possibles consiste à passer d'une configuration `●●○` à `○○●` et ainsi à diminuer d'un pion le nombre total de pions. Ces configurations peuvent être rencontrées dans les directions horizontales ou verticales.

On considère que la partie est gagnée quand il n'y a plus qu'un pion sur le plateau.

II.5.i Première implémentation naïve

On va représenter un plateau par le type OCaml :

```

OCaml | type case = Vide | Pion | Invalidé
      | type plateau = case array array
      | let n = 7
      | (* Un plateau est une matrice 7x7 avec des cases invalides aux coins *)
  
```

Question II.1 Définir une fonction `print_plateau` qui affiche un plateau sous un format textuel lisible.

Démonstration.

```

OCaml | let print_plateau p =
      |   for j = 0 to n-1 do
      |     for i = 0 to n-1 do
      |       let c = match p.(i).(j) with
      |         | Invalidé -> ' '
      |         | Vide -> '.'
      |         | Pion -> '#'
      |         in Printf.printf "%c" c
      |     done;
      |     print_newline ()
      |   done;
      |   print_newline ()
  
```

Question II.2 Écrire une fonction `plateau_initial : unit -> plateau` qui renvoie un plateau correspondant à la configuration de départ.

Démonstration.

```
OCaml | let plateau_initial () =
[|
  [| Invalide; Invalide; Pion; Pion; Pion; Invalide; Invalide |];
  [| Invalide; Invalide; Pion; Pion; Pion; Invalide; Invalide |];
  [| Pion; Pion; Pion; Pion; Pion; Pion; Pion |];
  [| Pion; Pion; Pion; Vide; Pion; Pion; Pion |];
  [| Pion; Pion; Pion; Pion; Pion; Pion; Pion |];
  [| Invalide; Invalide; Pion; Pion; Pion; Invalide; Invalide |];
  [| Invalide; Invalide; Pion; Pion; Pion; Invalide; Invalide |]
|]
```



Un mouvement peut être assimilé à un triplet de coordonnées décrivant, dans l'ordre ● ● ○ les trois cases concernées. Il se trouve que la case centrale est toujours le milieu des deux autres, on peut donc se contenter de donner un couple de coordonnées pour décrire un mouvement.

```
OCaml | type mouvement = (int * int) * (int * int)
```

Question II.3 Écrire une fonction `mouvements : plateau -> mouvement list` qui renvoie la liste des mouvements possibles sur le plateau passé en paramètre.



Démonstration.

```
OCaml | let possible p (i1,j1) (i3,j3) =
  let i2, j2 = (i1+i3)/2, (j1+j3)/2 in
  p.(i1).(j1) = Pion && p.(i2).(j2) = Pion && p.(i3).(j3) = Vide

let mouvements p =
  (* on utilise une pile *)
  let l = ref [] in
  for i = 0 to n-1 do
    for j = 0 to n-3 do
      if possible p (i,j) (i,j+2)
      then l := ( (i,j), (i,j+2) ) :: !l;
      if possible p (i,j+2) (i,j)
      then l := ( (i,j+2), (i,j) ) :: !l;
      if possible p (j+2,i) (j,i)
      then l := ( (j+2,i), (j,i) ) :: !l;
      if possible p (j,i) (j+2,i)
      then l := ( (j,i), (j+2,i) ) :: !l
    done
  done;
  !l
```



Question II.4 Écrire une fonction `compte_pions : plateau -> int` qui renvoie le nombre de pions sur un plateau.

QUESTION QUESTION

En déduire une fonction valide : plateau \rightarrow bool qui indique si un plateau correspond à une partie gagnante.

Démonstration.

```
OCaml
let compte_pions p =
  let c = ref 0 in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if p.(i).(j) = Pion
      then incr c
    done
  done;
  !c

let valide p = compte_pions p = 1
```

Question II.5 Écrire deux fonctions faire et defaire de type plateau \rightarrow mouvement \rightarrow unit permettant de faire et défaire un mouvement.

Démonstration.

```
OCaml
let applique p mouv =
  let (i1,j1), (i3,j3) = mouv in
  let i2, j2 = (i1+i3)/2, (j1+j3)/2 in
  p.(i1).(j1) <- Vide;
  p.(i2).(j2) <- Vide;
  p.(i3).(j3) <- Pion

let defaire p mouv =
  let (i1,j1), (i3,j3) = mouv in
  let i2, j2 = (i1+i3)/2, (j1+j3)/2 in
  p.(i1).(j1) <- Pion;
  p.(i2).(j2) <- Pion;
  p.(i3).(j3) <- Vide
```

Question II.6 Écrire une fonction enumere : plateau \rightarrow mouvement list \rightarrow unit telle que enumere pos chemin énumère les plateaux accessibles depuis pos jusqu'à obtenir une solution et sachant que chemin est la liste de mouvements, du plus récent au plus ancien, qui ont conduit jusqu'à pos.

En cas de succès, on produira une exception Solution of mouvement list renvoyant la liste des mouvements ayant conduits à une solution.

En déduire une fonction resout : unit \rightarrow mouvement list qui renvoie une liste de mouvement permettant de résoudre le solitaire. On prendra garde à renverser le chemin obtenu pour que le premier mouvement de la liste soit le premier mouvement effectué.

Démonstration.

Ocaml

```
exception Solution of mouvement list

let rec enumere pos chemin =
  if valide pos
  then raise (Solution chemin)
  else let l = mouvements pos in
    List.iter (fun mouv ->
      applique pos mouv;
      enumere pos (mouv :: chemin);
      defaire pos mouv) l

let resout () =
  let pos = plateau_initial () in
  try
    enumere pos [];
    raise Not_found
  with Solution l -> List.rev l
```

■

Ce code ne permet pas de calculer la solution car il prend beaucoup trop de temps en raison du nombre de positions étudiées.

II.5.ii Cache des mauvaises positions

On se rend compte que de nombreuses positions sont réétudiées alors qu'on sait déjà qu'elles ne peuvent permettre d'aboutir à une solution. En effet, il y a souvent des coups indépendants pouvant être joués au même moment, ce qui fait qu'on peut aboutir à une même position de beaucoup de manière différente, ce qui augmente exceptionnellement le nombre d'appels récursifs.

Une stratégie consiste à maintenir un ensemble de configurations mauvaises. Pour réaliser un tel ensemble, on va utiliser une table de hâchage dont les clés sont les positions et les valeurs `unit`. Si une position a une valeur associée dans la table, c'est qu'elle sera mauvaise.

Cela pose la question de la représentation persistante et immuable des positions. Une première stratégie peut consister à transformer le plateau en `case list list`. Cette stratégie est beaucoup trop coûteuse et elle ne permet pas de répondre instantanément. On va profiter du fait qu'il n'y est que 49 cases dans le plateau pour le représenter par un entier sur 49 bits : $a_{00} + a_{10}2 + a_{20}2^2 + \dots + a_{60}2^6 + a_{01}2^7 + \dots + a_{66}2^{48}$ où a_{ij} vaut 1 lorsqu'il y a un pion sur la j ème ligne et la i ème colonne, c'est-à-dire quand $p.(i).(j) = \text{Pion}$.

Indication l'entier 1 `lsl n` est 2^n . `lsl` signifie qu'on décale le chiffre 1 de n bits vers la droite dans son écriture binaire.

Question II.7 Écrire une fonction `code : plateau -> int` qui renvoie le numéro associé à un plateau.

■

Démonstration.

Ocaml

```
let code p =
  let c = ref 0 in
  for j = 0 to 6 do
    for i = 0 to 6 do
```

```

    if p.(i).(j) = Pion
    then c := !c + 1 lsl (7*j+i)
done
done;
!c

```



Pour manipuler un ensemble, on va définir

```

OCaml let mauvaises = Hashtbl.create 42
let ajoute x = Hashtbl.add mauvaises x ()
let contient x = Hashtbl.mem mauvaises x

```

L'appel à `ajoute x` rajoute `x` dans l'ensemble des mauvaises positions et `contient x` vérifie si `x` est dans cet ensemble.

Question II.8 Reprendre la fonction `enumere` avec un ensemble de mauvais codes.



Démonstration.

```

OCaml let rec enumere pos chemin =
  if valide pos
  then raise (Solution chemin)
  else
    let c = code pos in
    if not (contient c)
    then begin
      let l = mouvements pos in
      List.iter (fun mouv ->
                    applique pos mouv;
                    enumere pos (mouv :: chemin);
                    defaire pos mouv) l;
      (* Si on est ici c'est que le noeud ne permet pas de trouver
         une solution *)
      ajoute c
    end

```



Normalement, le code doit pouvoir permettre de réaliser la résolution instantanément.

■ **Remarque 14.4** Possibles extensions :

- rajouter un affichage de la résolution
- déterminer la proportion de recalculs évités
- gérer les symétries des plateaux



III Stratégies d'énumération

III.1 Combinatoire élémentaire

produits, combinaisons, permutations

III.2 Enumération d'arbres

Imaginons que l'on souhaite énumérer des arbres binaires non étiquetés pour trouver le premier arbre binaire à n noeuds vérifiant une certaine propriété. On suppose ainsi défini un type

```
OCaml | type arbre = Nil | Noeud of arbre * arbre
```

et une fonction pour tester le prédictat :

```
OCaml | val : valide : arbre -> bool
```

Une première possibilité est d'effectuer un simple parcours récursif :

```
OCaml | let rec recherche n =
  if n = 0
  then
```

normale, par passage de continuation

IV Droite de balayage

IV.1 Principe

Il est parfois possible d'ordonner V pour tirer pour permettre de trouver une solution plus vite, voir d'ordonner des données basique pour ne pas énumérer V mais énumérer un $V' \subset V$ plus petit.

C'est un procédé classique dans le contexte de la géométrie algorithmique dans le plan : étant donné un ensemble V de candidats qu'on déduit d'un ensemble de points du plan, par exemple l'ensemble des paires de points, on va énumérer V à l'aide d'un parcours des points de gauche à droite (ou tout autre direction géométrique) pour ne pas énumérer tout V mais seulement une partie plus petite. Tout se passe comme si on balayait avec un droite l'ensemble des points, d'où le nom de droite de balayage.

Du point de vue de la complexité temporelle, on obtient le plus souvent un algorithme en $O(n \log n)$ où n est le nombre de points. Cela signifie que l'étape la plus coûteuse en temps est le tri initial.

On va présenter ici deux exemples d'utilisation d'une droite de balayage. Il est assez clair qu'il sera nécessaire dans ces cas de réfléchir géométriquement. Les exemples présentés sont donc assez simples d'un point de vue informatique, mais plutôt complexes d'un point de vue mathématiques.

IV.2 Plus proche paire

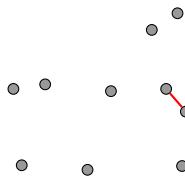
On considère le problème

Problème - PLUSPROCHEPAIRE

- Entrées :
 - Un ensemble P de n points dans le plan.
- Sortie : Une paire $\{p, p'\}$ de points telle que $dist(p, p') = ||\vec{pp'}||$ soit minimale.

IV.2.i Recherche exhaustive

Considérons le problème *PlusProchePaire* qui, étant donné un ensemble de n points ($n \geq 2$), détermine la paire constituée des deux points les plus proches.



Une implémentation naïve de la recherche par force brute consiste à énumérer les $\frac{n(n-1)}{2}$ paires et donc à effectuer $O(n^2)$ itérations.

```
let plus_proche_paire points =
  let n = Array.length points in
  let min_paire = ref (distance points.(0) points.(1), (0, 1)) in
  for i = 0 to n - 1 do
    for j = i+1 to n - 1 do
      let d = distance points.(i) points.(j) in
      if d < fst !min_paire
      then min_paire := (d, (i, j))
    done
  done;
  snd !min_paire
```

Ocaml

IV.2.ii Raffinement : droite de balayage

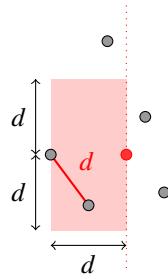
Il est parfois possible d'accélérer la recherche par force brute en ordonnant le parcours des candidats pour pouvoir éviter de tester certains d'entre eux.

En géométrie algorithmique, une approche classique consiste à ordonner les objets selon leur abscisse et à parcourir les objets par abscisse croissante. On parle alors de **droite de balayage** (en anglais, *sweep line*) car cela revient à balayer le plan par une droite verticale en ne traitant que les objets avant cette ligne.

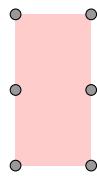
Reprenons le problème précédent, on considère que les points sont triés par abscisse croissante : $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$. On va parcourir les points dans cet ordre en maintenant un ensemble de points à gauche du point courant, appelés *points actifs*, et en ne calculant que les intersections avec les points actifs.

Si on a parcouru les N premiers points et qu'on a obtenu que la plus petite distance était d , lorsqu'on considère le point (x_N, y_N) , il est inutile de tester les points qui sont forcément à distance $> d$ de celui-ci. C'est-à-dire qu'on peut éliminer les points qui ne sont pas dans le rectangle $[x_N - d, x_N] \times [y_N - d, y_N + d]$ du test. Les points dont l'abscisse est $< x_N - d$ peuvent être éliminés définitivement vu que l'on raisonne par abscisse croissante, par contre, les points d'ordonnées invalides doivent être conservés pour les points ultérieurs.

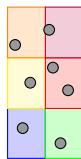
Ce rectangle est représenté sur le schéma suivant ainsi qu'une ligne imaginaire qui correspond à l'abscisse du point courant et qu'on peut imaginer parcourant le plan de gauche à droite pour traiter les points au fur et à mesure.



Afin de déterminer la complexité de cet algorithme, il est nécessaire de connaitre le nombre maximal de points dans le rectangle. Comme ces points ont été pris en compte précédemment, ils sont forcément à distance au moins d les uns des autres. Il s'agit donc de déterminer le nombre maximum de points qu'on peut placer dans ce rectangle à distance au moins d . On remarque tout d'abord qu'on peut placer six points ainsi :



Si jamais on avait au moins sept points, on peut voir qu'il y a forcément un des six sous-rectangles suivants qui contiendrait au moins deux points :



Or, ces sous-rectangles sont de longueur $\frac{1}{2}d$ et de hauteur $\frac{2}{3}d$, donc la distance maximale entre deux de leurs points correspond à la longueur des diagonales : $\sqrt{\frac{1}{4} + \frac{4}{9}}d = \frac{5}{6}d < d$.

Comme un de ces six points est le point courant, il y a toujours au plus 5 points dans l'ensemble des points actifs.

Voici le principe de l'algorithme que l'on va implémenter :

- On trie le tableau `points` par ordre croissant. **Complexité** : $O(n \log n)$
- On initialise la plus petite distance d courante à la distance entre les deux premiers points
- On crée un ensemble `actifs`, ordonné par les ordonnées, de points contenant initialement les deux premiers points
- Pour chaque point (x, y) en partant du deuxième :
 - ★ On supprime les points (x', y') tels que $x' < x - d$ de `actifs`. **Complexité** : sur l'ensemble des itérations on ne pourra jamais supprimer deux fois un point, donc on effectue au maximum n suppressions chacune en $O(\log n)$ donc $O(n \log n)$.
 - ★ On parcourt les points de `actifs` dont les ordonnées sont comprises entre $y - d$ et $y + d$. **Complexité** : pour récupérer le premier point de l'ensemble, il faut $O(\log n)$ en pire cas (tous les points actifs) et ensuite on effectue au plus 5 itérations comme on vient de le prouver.

On remarque ainsi que la complexité en temps et en pire cas de cet algorithme est de $O(n \log n)$. Ici, le fait d'avoir la structure `actifs` ordonnée par les ordonnées est crucial pour garantir la complexité. Pour la réalisation d'une structure d'ensemble ordonnée ayant ces complexités, voir le chapitre FIXME.

Ici, on utilise le module Set d'OCaml pour réaliser la structure d'ensemble, pour cela on commence par créer le module PointSet pour les ensembles de points :

```
OCaml
module Point = struct
  type t = float * float
  let compare (x1,y1) (x2,y2) = Stdlib.compare y1 y2
end

module PointSet = Set.Make(Point)
```

Puis on définit une fonction permettant de parcourir les points entre deux ordonnées :

```
OCaml
let set_iter_entre f set bas haut =
  try
    let e = PointSet.find_first (fun p -> snd p >= bas) set in
    let seq = PointSet.to_seq_from e set in
    let rec aux seq =
      match seq () with
      | Seq.Nil -> ()
      | Seq.Cons (p, seq_suite) ->
        if snd p <= haut
        then begin
          f p;
          aux seq_suite
        end
    in aux seq
  with Not_found -> ()
```

On implémente alors assez directement l'algorithme décrit précédemment :

```
OCaml
let plus_proche_paire_balayage points =
  let compare (x1,y1) (x2,y2) =
    if x1 = x2
    then if y1 < y2 then -1 else 1
    else if x1 < x2 then -1 else 1
  in
  Array.sort compare points;
  let n = Array.length points in
  let d = ref (distance points.(0) points.(1)) in
  let couple = ref (points.(0), points.(1)) in
  let actifs = ref (PointSet.empty
    |> PointSet.add points.(0) |> PointSet.add points.(1)) in

  let gauche = ref 0 in

  for i = 2 to n-1 do
    let xi, yi = points.(i) in

    while fst points.(!gauche) < xi -. !d do
      actifs := PointSet.remove points.(!gauche) !actifs;
      incr gauche
    done;

    set_iter_entre (fun pj ->
      let dip = distance points.(i) pj in
```

```

if dip < !d
then begin
    couple := (points.(i), pj);
    d := dip
end) !actifs (yi -. !d) (yi +. !d);

actifs := PointSet.add points.(i) !actifs
done;
!d

```

IV.2.iii Problème : test d'intersection pour un ensemble de segments

Considérons le problème suivant *IntersectionEnsemble* : étant donné n segments dans le plan, il s'agit de déterminer si au moins deux des segments s'intersectent.

■ **Remarque 14.5** On peut considérer ici que l'on dispose d'une fonction

Ocaml `intersecte : (float * float) * (float * float)
-> (float * float) * (float * float) -> bool`

qui teste l'intersection entre deux segments.

Cependant, il est possible d'écrire une telle fonction avec un peu de géométrie élémentaire.

Si on considère que les deux segments sont $[A_1B_1]$ et $[A_2B_2]$, avec $A_1 \neq B_1$ et $A_2 \neq B_2$, alors chaque point du segment $[A_1B_1]$ est de la forme $A_1 + t\overrightarrow{A_1B_1}$ où $t \in [0, 1]$. De même les points du segment $[A_2B_2]$ sont de la forme $A_2 + u\overrightarrow{A_2B_2}$ où $u \in [0, 1]$.

S'il y a une intersection, c'est qu'il existe $(t, u) \in [0, 1]^2$ tel que

$$A_1 + t\overrightarrow{A_1B_1} = A_2 + u\overrightarrow{A_2B_2} \iff \overrightarrow{A_2A_1} + t\overrightarrow{A_1B_1} = u\overrightarrow{A_2B_2}$$

L'idée est alors d'utiliser une opération appelée **produit vectoriel** sur les vecteurs. Comme ici, tout est plan, le produit vectoriel est uniquement déterminé par sa troisième coordonnée, celle qui sort du plan, et on peut se contenter de calculer celle-ci. On note ainsi $(x, y) \times (x', y') = xy' - yx'$ cette coordonnée. On a donc $u \times u = 0$.

On peut alors composer l'égalité par $\times \overrightarrow{A_2B_2}$:

$$\overrightarrow{A_2A_1} \times \overrightarrow{A_2B_2} + t (\overrightarrow{A_1B_1} \times \overrightarrow{A_2B_2}) = 0$$

Notons $\Delta = \overrightarrow{A_1B_1} \times \overrightarrow{A_2B_2}$, si $\Delta \neq 0$, alors

$$t = -\frac{\overrightarrow{A_2A_1} \times \overrightarrow{A_2B_2}}{\Delta} = \frac{\overrightarrow{A_1A_2} \times \overrightarrow{A_2B_2}}{\Delta}$$

On procède de même avec $\times \overrightarrow{A_1B_1}$ pour obtenir une expression de u : $\overrightarrow{A_2A_1} \times \overrightarrow{A_1B_1} = u (\overrightarrow{A_2B_2} \times \overrightarrow{A_1B_1}) = -u\Delta$ et donc

$$u = -\frac{\overrightarrow{A_2A_1} \times \overrightarrow{A_1B_1}}{\Delta} = \frac{\overrightarrow{A_1A_2} \times \overrightarrow{A_1B_1}}{\Delta}$$

Si $\Delta \neq 0$, on peut donc alors exprimer u et t et vérifier qu'ils sont dans $[0, 1]$.

Si $\Delta = 0$ c'est que les deux segments sont de directions parallèles ou confondues.

- Si $\overrightarrow{A_1A_2} \times \overrightarrow{A_1B_1} \neq 0$ alors $\overrightarrow{A_1A_2}$ et $\overrightarrow{A_1B_1}$ sont non colinéaires donc les deux segments sont sur des droites parallèles distinctes et ne peuvent s'intersecter.
- Sinon, les segments reposent sur une même droite et il s'agit de vérifier leurs positions sur la droite. Pour cela, on exprime $A_2 = A_1 + t_A \overrightarrow{A_1B_1}$ de même pour $B_2 = A_1 + t_B \overrightarrow{A_1B_1}$. Plus précisément, on calcule $\overrightarrow{A_1A_2} \cdot \overrightarrow{A_1B_1} = t_A \|\overrightarrow{A_1B_1}\|^2$ à l'aide du produit scalaire et on a $t_A = \frac{\overrightarrow{A_1A_2} \cdot \overrightarrow{A_1B_1}}{\|\overrightarrow{A_1B_1}\|^2}$. De même, $t_B = \frac{\overrightarrow{A_1B_2} \cdot \overrightarrow{A_1B_1}}{\|\overrightarrow{A_1B_1}\|^2}$. On doit alors vérifier si l'intervalle $[t_A, t_B]$ (ou $[t_B, t_A]$ selon leur position) intersecte $[0, 1]$.

Voici une fonction OCaml qui correspond à ce raisonnement

```
let intersecte (a1,b1) (a2,b2) =
  let vec (x1,y1) (x2,y2) = (x2-.x1,y2-.y1) in
  let cross (x1,y1) (x2,y2) = x1 *. y2 -. y1 *. x2 in
  let dot (x1,y1) (x2,y2) = x1 *. x2 +. y1 *. y2 in
  let proche0 x = let eps = 1e-20 in
    if x < 0. then -x < eps else x < eps in
  let a1b1 = vec a1 b1 in let a2b2 = vec a2 b2 in
  let a1a2 = vec a1 a2 in let a1b2 = vec a1 b2 in

  let delta = cross a1b1 a2b2 in

  if proche0 delta
  then
    if proche0 (cross a1a2 a1b1)
    then let na1b1 = dot a1b1 a1b1 in (* colinéaires *)
        let tA = (dot a1a2 a1b1) /. na1b1 in
        let tB = (dot a1b2 a1b1) /. na1b1 in
        if tA < tB
        then not (tB < 0. || tA > 1.)
        else not (tA < 0. || tB > 1.)
    else false (* parallèles *)
  else let t = (cross a1a2 a2b2) /. delta in (* se croisent *)
        let u = (cross a1a2 a1b1) /. delta in
        t >= 0. && t <= 1. && u >= 0. && u <= 1.
```

OCaml

■ **Note 14.1** réécrire cela avec le déterminant de deux vecteurs du plan qui est au programme de mathématiques de seconde.

La recherche par force brute va alors énumérer l'ensemble des paires de segments distincts et tester deux à deux les intersections. On peut ainsi écrire le programme suivant qui est assez simple et effectuera effectivement $O(|v|^2)$ itérations dans le pire cas, i.e. lorsqu'il n'y a pas d'intersections.

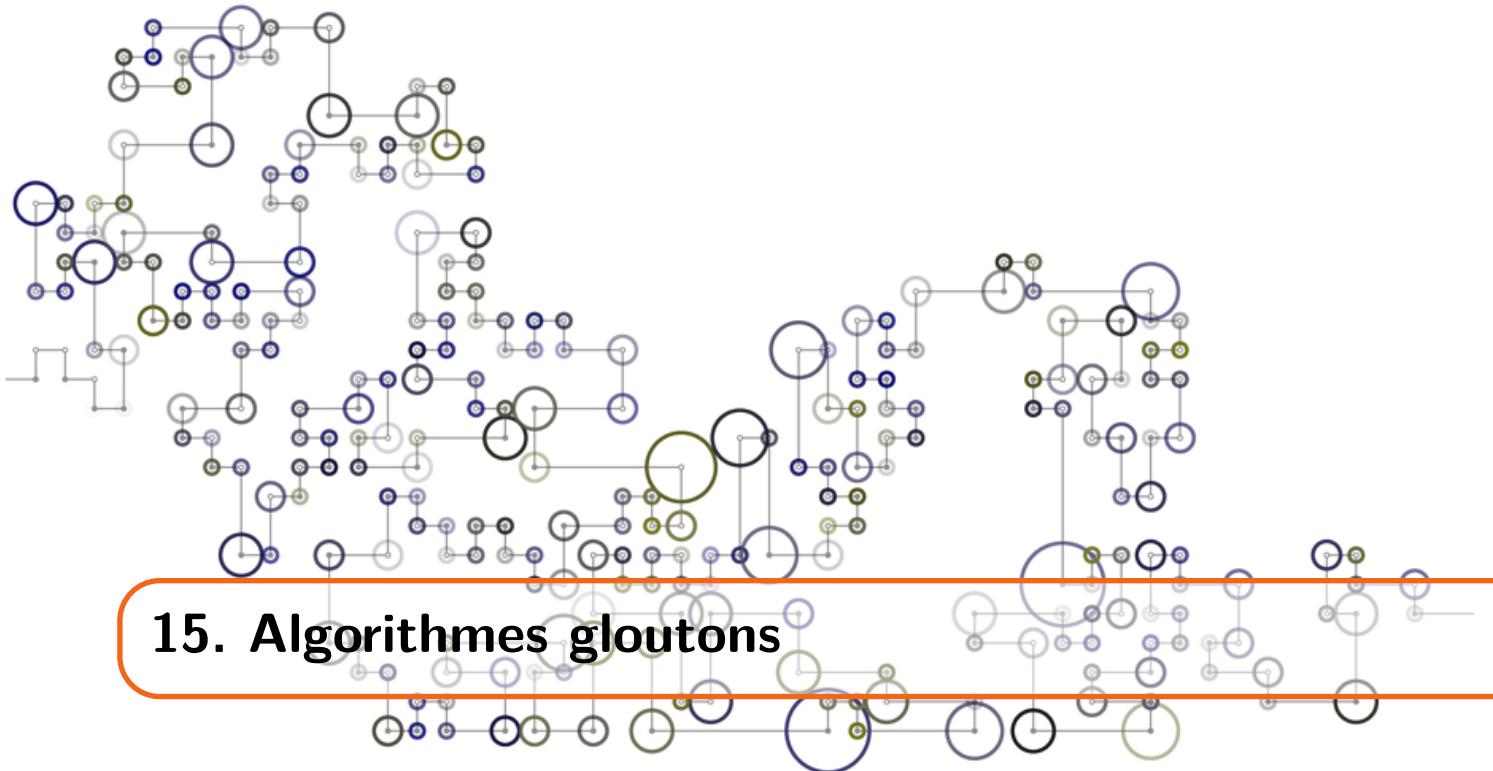
```
exception Trouve

let intersection_ensemble (v: ((float * float) * (float * float)) array) : bool =
  let n = Array.length v in
  try
    for i = 0 to n - 1 do
      for j = i+1 to n-1 do
```

OCaml

```
    if intersecte v.(i) v.(j)
    then raise Trouve
done
done;
false
with Trouve -> true
```

TODO approche par droite de balayage : algorithme de Shamos et Hoey (1976)



15. Algorithmes gloutons

I Principes

1.1 Problème d'optimisation combinatoire

On considère ici un problème d'énumération comme dans le chapitre sur le retour sur trace : un ensemble **fini** S muni d'une valeur $s_0 \in S$ appelée la **position** initiale et de deux fonctions

$$mouv : S \rightarrow \mathcal{P}(S)$$

$$taille : S \rightarrow \mathbb{N}$$

telles que $taille(s_0) = 0$ et que

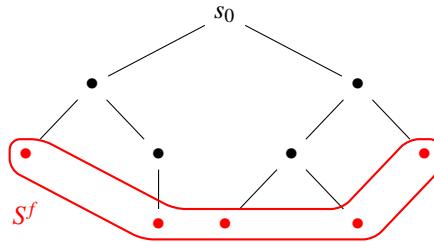
$$\forall x, y \in S, x \in mouv(y) \Rightarrow taille(x) < taille(y)$$

Cette dernière propriété assure qu'en partant de s_0 et en effectuant une série de *mouvements* on aboutit forcément à une position finale, c'est-à-dire pour laquelle $mouv(s) = \emptyset$. On note $S^f \subset S$ les positions finales.

On peut donc ainsi tracer l'arbre des positions où s_0 est à la racine et les enfants d'une position sont celles données par $mouv$. Attention, dans cet arbre, on peut distinguer deux positions selon les mouvements qui nous y amènent. En fait, si on partage ces positions, il s'agit d'un graphe acyclique non orienté.

Exercice 15.1 Prouver cette affirmation.

Sur la représentation suivante, on a fait figurer en rouge les éléments de S^f .



Dans le retour sur trace, on cherchait une position finale vérifiant une certaine propriété. Ici, on considère une fonction objectif :

$$\chi : S^f \rightarrow \mathbb{R}$$

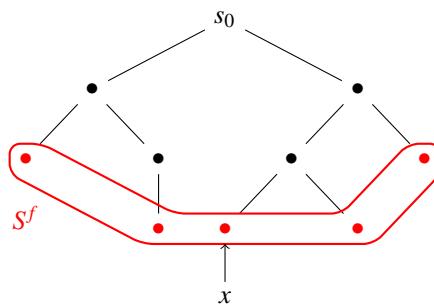
qui associe une valeur numérique à chaque position. On considère un problème dit de maximisation : déterminer $x \in S$, tel que $\chi(x) = \max_{y \in S} \chi(y)$. On notera souvent $x = \operatorname{argmax}_{y \in S} \chi(y)$. Quand le problème considéré dépend d'une entrée E , on note $\text{Opt}(E)$ la valeur de l'optimal.

■ **Remarque 15.1**

- En considérant $g : y \mapsto -f(y)$, on transforme un problème de maximisation en un problème de minimisation. On parle plus généralement de problèmes d'optimisation combinatoire.
- Il y a une ambiguïté sur $\operatorname{argmax}_{y \in S} f(y)$ quand plusieurs éléments de S réalisent ce maximum. Dans la plupart des algorithmes gloutons qu'on va considérer, on commence par donner un ordre sur S et on considère le plus petit y pour cet ordre réalisant le maximum. L'ordre choisi est alors crucial dans la preuve de correction. C'est aussi une des raisons pour lesquelles les algorithmes gloutons sont souvent de complexité temporelle $O(n \log_2 n)$.

Une première stratégie très élémentaire consiste alors à énumérer S , de manière exhaustive ou avec une stratégie plus fine comme le retour sur trace, puis à déterminer un élément maximal de manière directe.

Cela revient donc à déterminer l'arbre des solutions puis à trouver une feuille maximisant l'objectif :



1.2 Algorithme glouton

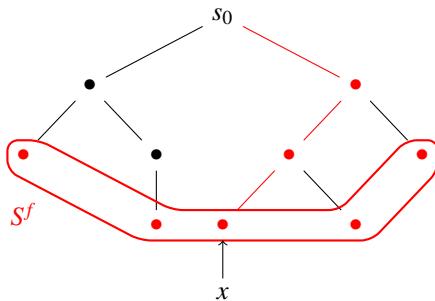
Le nombre de nœud de l'arbre est en général exponentiel en la taille de l'entrée. Ainsi, une exploration exhaustive est optimale mais très coûteuse.

Un algorithme glouton va suivre une approche beaucoup plus efficace : à chaque étape de construction de la solution, on choisit la branche qui maximise la fonction d'objectif. Pour cela, on considère que la fonction d'objectif χ se prolonge à toutes les positions dans S .

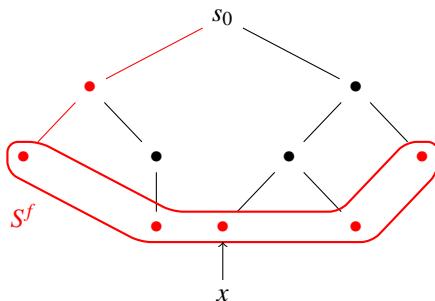
Si la position courante est $x \in S$, la position suivante sera $y = \operatorname{argmax}_{z \in mouv(x)} \chi(z)$.

On dit que y est le **mouvement glouton** ou **choix glouton**.

Sur l'arbre précédent, cela reviendrait à n'emprunter qu'une seule branche indiquée en rouge sur le schéma.



Cela a l'air très efficace mais il y a un problème majeur : il n'y a aucune garantie qu'on aboutisse ainsi à une solution, encore moins à une solution optimale. En effet, on aurait très bien pu faire les choix suivants :



et ne pas aboutir à une solution.

1.3 Cas du rendu de monnaie

Considérons par exemple le problème du **rendu de monnaie** : étant donné, une liste de valeurs faciales de pièces $P = (v_1, \dots, v_p) \in (\mathbb{N}^*)^p$ avec $1 = v_1 < \dots < v_p$ et une somme $n \in \mathbb{N}^*$, on cherche la manière d'exprimer cette somme avec le plus petit nombre de pièces possible.

Plus précisément, l'ensemble des positions est $S = \{(k_1, \dots, k_p) \in \mathbb{N}^p \mid k_1v_1 + \dots + k_pv_p \leq n\}$, les positions finales est $S^f = \{(k_1, \dots, k_p) \in \mathbb{N}^p \mid k_1v_1 + \dots + k_pv_p = n\}$ et la fonction d'objectif est $\chi : (k_1, \dots, k_p) \mapsto k_1 + \dots + k_p$.

On cherche alors $x = \operatorname{argmin}_{y \in S^f} \chi(y)$.

Comme $1 = v_1$, $S \neq \emptyset$ car $(n, 0, \dots, 0) \in S$ et ainsi $\chi(x) \leq n$.

■ **Exemple 15.1** Si $P = (1, 2, 5, 10)$ et $n = 14$ on a $(3, 3, 1, 0) \in S^f$ car $n = 3 + 3 \times 2 + 1 \times 5$. ■

L'algorithme glouton va utiliser la plus grande pièce possible à chaque étape : si on doit décomposer n , on cherche $p_i \leq n$ maximale, on décompose $n - p_i$ et on rajoute la pièce i pour en déduire une décomposition de n . On s'arrête quand le nombre à décomposer vaut 0 qui correspond à l'unique rendu vide.

■ **Exemple 15.2** Si $P = (1, 2, 5, 10)$ et $n = 14$.

- On utilise la plus grande pièce possible $10 \leq 14$ puis on exprime $4 = 14 - 10$
- Ici, la plus grande pièce est 2 et on continue avec $2 = 4 - 2$
- La plus grande pièce est encore 2 et on s'arrête car $0 = 2 - 2$.

- En conclusion, on a obtenu $x = (0, 2, 0, 1)$.
- Une exploration exhaustive permet de s'assurer qu'on a effectivement obtenu une décomposition minimale. En effet, ici l'ensemble des décompositions est : $\{(14,0,0,0), (12,1,0,0), (8,3,0,0), (6,4,0,0), (4,5,0,0), (2,6,0,0), (0,7,0,0), (9,0,1,0), (7,1,1,0), (5,2,1,0), (3,3,1,0), (1,4,1,0), (4,0,2,0), (2,1,2,0), (0,2,2,0), (4,0,0,1), (2,1,0,1), (0,2,0,1)\}$. ■

■ **Exemple 15.3** On considère maintenant $P = (1, 2, 7, 10)$ et $n = 14$

- L'algorithme glouton va ici procéder comme dans l'exemple précédent et on va obtenir $x = (0, 2, 0, 1)$.
- Mais on remarque que ce n'est pas un minimum car $x' = (0, 0, 2, 0)$ convient avec $f(x') = 2 < 3 = f(x)$. ■

■ **Remarque 15.2** L'algorithme glouton n'est pas nécessairement optimal. Un système de pièces, on parle de système monétaire, telle que l'algorithme glouton soit optimal est appelé un *système canonique*. Un exercice en fin de chapitre demande de démontrer qu'un certain système est canonique. ■

On peut se poser la question des algorithmes pour lesquels l'algorithme glouton aboutit nécessairement à une solution optimale. Il se trouve qu'il y a un cadre général mais hors programme dont on pourra voir se dégager la nature à la lumière des preuves d'optimalité qui vont suivre.

II Construction de l'arbre de Huffman

II.1 Description

■ **Remarque 15.3** Ce paragraphe décrit l'étape cruciale du principe de compression de Huffman. Celui-ci sera présenté complètement dans le chapitre Algorithmique des textes. ■

On va étudier ici un principe de compression parfaite (sans perte d'information à la décompression) de données appelé l'algorithme de Huffman et qui repose sur ce principe simple : coder sur moins de bits les caractères les plus fréquents.

Par exemple si on considère le mot abaabc, en le codant avec un nombre de bits fixes, par exemple 2 avec le code a=00, b=01, c=10, on aurait besoin de 12 bits pour représenter le mot. Mais si on choisit le code suivant : a=0, b=10, c=11, il suffit de 9 bits. On a donc gagné 3 bits soit un facteur de compression de 75%.

On remarque que pour pouvoir décompresser, il n'aurait pas été possible de faire commencer le code de b ou c par un 0, sinon on aurait eu une ambiguïté avec la lecture d'un a. On parle alors de code préfixe :

Définition II.1 Soit $X \subset \{0, 1\}^*$, on dit que X est un code préfixe lorsque pour tous $x, y \in X$, x n'est pas un préfixe de y et y n'est pas un préfixe de x .

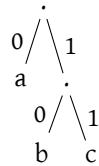
On se pose alors la question du code préfixe optimal pour un texte donné.

Plus précisément, étant donné un alphabet fini Σ et une application $f : \Sigma \rightarrow [0, 1]$ associant à chaque lettre son nombre d'occurrences dans le texte considéré. Ainsi $\sum_{x \in \Sigma} f(x)$ est la longueur du texte. On cherche un code préfixe X et une application $c : \Sigma \rightarrow X$ telle que $\sum_{x \in \Sigma} f(x)|c(x)|$ soit minimale car cela correspond au nombre de bits après codage.

■ **Remarque 15.4** On utilise aussi la notion de fréquence du lettre qui est son nombre d'occurrence rapporté à la longueur du texte. Un des avantages de la notion de fréquence est qu'il est possible de considérer une table de fréquence déjà construite comme celle de la langue française.

L'application de codage c peut être représenté par un arbre binaire où les arêtes gauches correspondent à 0, les arêtes droites à 1 et les feuilles aux éléments de Σ dont les étiquettes des chemins y menant depuis la racine de l'arbre correspondent à leur image par c .

Par exemple, pour le code $a = 0, b = 10, c = 11$ on aurait l'arbre :



Avec un tel arbre, il est très simple de décoder le texte codé car il suffit de suivre un chemin dans l'arbre jusqu'à tomber sur une feuille, produire la lettre correspondante, puis repartir de la racine de l'arbre. La longueur du code associé à une lettre est alors égale à la profondeur de la feuille correspondante. L'optimalité du codage préfixe est ainsi équivalente à la minimalité de l'arbre vis-à-vis de la fonction d'objectif $\varphi(t) = \sum_{x \in \Sigma} f(x)p(t, x)$ où $p(t, x)$ est la profondeur de la feuille d'étiquette x dans l'arbre t ou 0 si x n'est pas une des étiquettes, cet extension permettant d'étendre la fonction d'objectif aux solutions partielles.

II.2 Algorithme glouton et implémentation

L'algorithme d'Huffman va construire un arbre correspondant à un codage optimal à l'aide d'une file de priorité d'arbres. On étend pour cela l'application f à de tels arbres en définissant que si t est un arbre de feuilles x_1, \dots, x_n alors $f(t) = f(x_1) + \dots + f(x_n)$.

- Au départ, on place dans la file des arbres réduits à une feuille pour chaque élément $x \in \Sigma$ et dont la priorité est $f(x)$.
- Tant que la file contient au moins deux éléments
 - ★ on retire les deux plus petits éléments x et y de la file de priorité $f(x)$ et $f(y)$
 - ★ on ajoute un arbre $z = \text{Noeud}(x, y)$ de priorité $f(z) = f(x) + f(y)$.
- On renvoie l'unique élément restant dans la file.

L'implémentation de cet algorithme est alors assez directe avec une file de priorité. On réutilise ici la structure de tas implementée en FIXME. Comme il s'agit d'un tas max, on insère avec $-f(x)$ comme valeur.

ERROR: `src/algorithme/.../.../snippets/algorithme/huffman.c` does not exist

L'algorithme de Huffman est un algorithme glouton car si on considère pour solution partielle la fôret présente dans la file et pour objectif la fonction φ étendue aux fôrets en sommant la valeur de φ sur chaque arbre, alors fusionner dans la fôret F deux arbres x et y en la transformant en une fôret F' va avoir l'impact suivant sur la fonction d'objectif :

$$\varphi(F') = \varphi(F) + f(x) + f(y)$$

car, en effet, on va rajouter 1 à la profondeur de chaque feuille et donc on passe pour la contribution de x de $\varphi(x) = \sum_{c \in \Sigma} f(c)p(x, c)$ à $\sum_{c \in \Sigma} f(c)(p(x, c) + 1) = \varphi(x) + \sum_{c \in \Sigma} f(c) = \varphi(x) + f(x)$.

On remarque ainsi que la fusion qui minimise localement φ est celle qui fusionne les deux arbres de plus petite valeur pour f .

II.3 Preuve d'optimalité

Pour montrer que l'algorithme glouton produit ici un codage minimal, on va utiliser une technique classique qui consiste à montrer qu'étant donné une solution optimale, on peut toujours la transformer sans augmenter sa valeur pour obtenir, de proche en proche, la solution renvoyée par le glouton.

Théorème II.1 Supposons que les lettres les moins fréquentes soient a et b , il existe un arbre optimal dont les deux feuilles étiquetées par a et b descendent du même noeud et sont de profondeur maximale.

Démonstration.

Considérons un arbre optimal t et soient c l'étiquette d'une feuille de profondeur maximale. On remarque qu'elle a forcément une feuille sœur car sinon, on pourrait omettre le noeud et l'arbre obtenu serait de plus petite valeur par φ .

FIXME : dessin

Soit d l'étiquette de cette feuille sœur. Sans perte de généralités, on suppose que $f(c) \leq f(d)$ et $f(a) \leq f(b)$. Comme a a le plus petit nombre d'occurrences, a $f(a) \leq f(c)$ et comme b est la deuxième, on a $f(b) \leq f(d)$. De plus, $p(t, a) \geq p(t, c)$ et $p(t, b) \geq p(t, d)$.

Si on échange les étiquettes a et c , seule les termes associées à ces lettres changent dans l'évaluation de φ . Si on note t' le nouvel arbre obtenu après cet échange, on a

$$\varphi(t') = \varphi(t) - f(a)p(t, a) - f(c)p(t, c) + f(a)p(t, c) + f(c)p(t, a)$$

Or, $f(c) \geq f(a)$ et $p(t, a) \geq p(t, c)$ donc

$$\varphi(t') = \varphi(t) + (f(c) - f(a))(p(t, a) - p(t, c)) \leq \varphi(t)$$

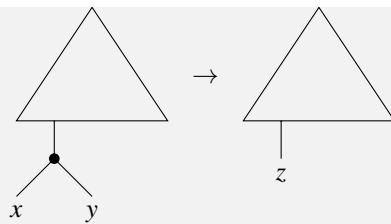
L'échange préserve le caractère optimal. En fait, ici, on a nécessairement une égalité pour ne pas aboutir à une contradiction, donc soit les feuilles étaient à même profondeur, soit les lettres avaient le même nombre d'occurrences.

Comme on a les mêmes relations entre b et d , on peut effectuer le même argument et échanger les étiquettes en préservant le caractère optimal.

■

Le théorème suivant permet de raisonner par récurrence en diminuant le nombre de lettres.

Théorème II.2 Soit t un arbre ayant x et y comme feuilles soeurs et t' l'arbre obtenu en remplaçant le noeud liant x et y par une feuille étiquetée par z où z est une nouvelle lettre telle que $f(z) = f(x) + f(y)$.



On a alors $\varphi(t) = \varphi(t') + f(z)$.

Démonstration.

Seule les termes portant sur x, y et z sont influencés par le changement et on a :

$$\begin{aligned}\varphi(t) &= \varphi(t') + f(x)p(t, x) + f(y)p(t, y) - f(z)p(t', z) \\ &= \varphi(t') + f(z)(p(t', z) + 1) - f(z)p(t', z) \\ &= \varphi(t') + f(z)\end{aligned}$$

■

Théorème II.3 L'algorithme de Huffman renvoie un arbre optimal.

Démonstration.

Par récurrence sur $|\Sigma|$.

Initialisation : si Σ ne contient qu'une lettre, il n'y a qu'un arbre qui est nécessairement optimal.

Héritéité : si la propriété est vraie pour un alphabet de $n - 1 \geq 1$ lettres, alors soit Σ contenant n lettres et x et y les deux lettres les moins fréquentes.

On pose Σ' obtenue en remplaçant x et y par une nouvelle lettre z et on suppose que $f(z) = f(x) + f(y)$. L'hypothèse de récurrence assure qu'on obtient un arbre optimal t' en appliquant l'algorithme d'Huffman sur Σ' . Comme la première étape d'Huffman va fusionner les feuilles x et y , on sait que l'arbre t obtenu en partant de Σ se déduit de t' en remplaçant z par $Noeud(x, y)$. Le théorème précédent assure alors que $\varphi(t) = \varphi(t') + f(z)$.

Soit t_o un arbre optimal pour Σ dans lequel x et y sont soeurs, possible en vertu du premier théorème, et soit t'_o l'arbre obtenu en remplaçant dans t_o le noeud liant x et y par une feuille étiquettée par z . On a ici encore $\varphi(t_o) = \varphi(t'_o) + f(z) \geq \varphi(t') + f(z) \geq \varphi(t)$ car t' est optimal.

Ainsi, on a bien l'égalité $\varphi(t_o) = \varphi(t)$ et t est optimal.

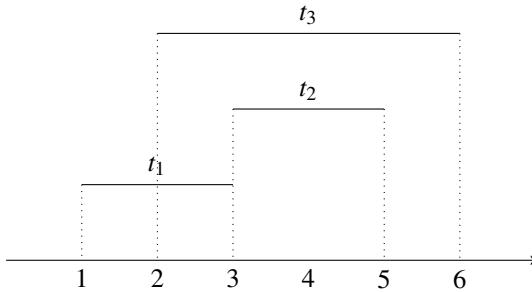
■

III Sélection d'activités

III.1 Description

Étant donné un ensemble d'activités données par leur temps de début et leur temps de fin (on considère les temps comme des entiers pour simplifier), on se pose la question du nombre maximal d'activité que l'on puisse sélectionner sans que deux activités soient en conflits. Cela correspond par exemple à l'organisation du planning d'un employé.

On dit que deux activités (d_1, f_1) et (d_2, f_2) sont en conflits quand $[d_1, f_1] \cap [d_2, f_2] \neq \emptyset$.



Ici, t_1 et t_2 sont en conflits avec t_3 . Mais t_1 et t_2 ne sont pas en conflit. On considère que deux activités peuvent se succéder directement : $f_1 = d_2$.

On considère donc en entrée de ce problème une suite finie $((d_1, f_1), \dots, (d_n, f_n))$ et on cherche un sous-ensemble $I \subset \llbracket 1, n \rrbracket$ de plus grand cardinal tel que pour tous $i, j \in I$, si $i \neq j$ alors (d_i, f_i) et (d_j, f_j) ne sont pas en conflits. On dit que I est un **ensemble indépendant**.

III.2 Algorithme glouton et implémentation

Pour résoudre ce problème, on considère l'algorithme glouton associé à la fonction d'objectif cardinal et **en triant les activités** ordre croissant de temps de fin.

Cet algorithme est implémenté dans le programme suivant :

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    unsigned int id;
    unsigned int debut;
    unsigned int fin;
    unsigned char selectionnee;
} activite;

int compare_activites(const void *t1, const void *t2)
{
    return ((activite *)t1)->fin - ((activite *)t2)->fin;
}

void selectionne(activite *activites, size_t nb_activites)
{
    size_t derniere_activite = 0;
    /* on commence par trier en  $O(n \log_2 n)$  les activités
     * selon le temps de fin */
    qsort(activites, nb_activites,
          sizeof(activite), compare_activites);
    activites[0].selectionnee = 1;
    for (size_t i = 1; i < nb_activites; i++)
    {
        if (activites[i].debut >= activites[derniere_activite].fin)
        {
            activites[i].selectionnee = 1;
            derniere_activite = i;
        }
    }
}
```

```

int main()
{
    activite activites[] = {
        { 0, 1, 3, 0 }, { 1, 3, 4, 0 }, { 2, 2, 5, 0 },
        { 3, 5, 9, 0 }, { 4, 11, 12, 0 }, { 5, 8, 10, 0 },
        { 6, 0, 7, 0 }
    };

    size_t nb_activites = sizeof(activites) / sizeof(activite);

    selectionne(activites, nb_activites);

    for (size_t i = 0; i < nb_activites; i++)
    {
        printf("Activité %d (%d,%d) : %d\n",
               activites[i].id, activites[i].debut,
               activites[i].fin, activites[i].selectionnee);
    }

    return 0;
}

```

Ce programme produit alors la sortie :

Activité 0 (1,3) : 1
 Activité 1 (3,4) : 1
 Activité 2 (2,5) : 0
 Activité 6 (0,7) : 0
 Activité 3 (5,9) : 1
 Activité 5 (8,10) : 0
 Activité 4 (11,12) : 1

■ **Remarque 15.5** Comme l'algorithme commence par effectuer un tri, on a rajouté dans la structure `activite` un champ permettant d'identifier une activité autrement que par son indice.

III.3 Preuve d'optimalité

On va prouver que l'algorithme glouton renvoie un ensemble indépendant optimal. Le fait que l'ensemble soit indépendant étant direct, on se concentre sur la preuve d'optimalité en présentant un schéma de preuve qui correspond à celui identifié dans le paragraphe précédent.

Théorème III.1 Si a_1, \dots, a_n sont des activités énumérées dans l'ordre croissant de leur temps de fin, alors il existe un ensemble indépendant optimal contenant a_1 .

■ **Remarque 15.6** Cela signifie qu'il fait le même choix que l'algorithme glouton à la première étape.

Démonstration.

Soit I un ensemble indépendant optimal ne contenant pas $a_1 = (d_1, f_1)$ (sinon c'est direct).

Si $a_k = (d_k, f_k)$ est l'activité de plus petit indice dans I , alors $f_k \geq f_1$ donc pour tout $a_i = (d_i, f_i)$ dans $I' = I \setminus \{a_k\}$ on a $d_i \geq d_k \geq f_1$ et ainsi a_1 et a_i ne sont pas en conflit. Ainsi $I' \cup \{a_1\}$ est un ensemble indépendant contenant a_1 de même cardinal que I donc optimal. ■

Théorème III.2 Soit $A = \{a_1, \dots, a_n\}$ des activités ordonnées par ordre croissant de temps de fin et I un ensemble indépendant optimal contenant $a_1 = (d_1, f_1)$ (ce qui est possible selon le théorème précédent).

$I' = I \setminus \{a_1\}$ est optimal pour $A' = \{(d, f) \in A \mid d \geq f_1\}$.

Démonstration.

Si, par l'absurde, I' est pas optimal pour A' alors $J \subset A'$ est un ensemble indépendant de cardinal strictement plus grand que celui de I' . Or, $A' \cup \{a_1\}$ est indépendant pour l'ensemble des activités et est de cardinal strictement plus grand que I . Contradiction. ■

Théorème III.3 L'algorithme glouton renvoie un ensemble indépendant optimal.

Démonstration.

Par récurrence forte sur le nombre d'activités.

- Initialisation : Pour une activité a_1 , le glouton renvoie $\{a_1\}$ qui est directement optimal.
- Hérédité : Si la propriété est vérifiée pour $k \leq n$ activités, soit $A = \{a_1, \dots, a_n\}$ des activités ordonnées par temps de fin. Soit I un ensemble indépendant optimal contenant a_1 et $I' = I \cap \{a_1\}$. Le théorème précédent assure que I' est optimal sur $A' = \{(d, f) \in A \mid d \geq f_1\}$.

Par hypothèse de récurrence, l'algorithme glouton sur A' produit un ensemble indépendant optimal G' , donc tel que $|G'| = |I'|$. Par construction l'algorithme glouton sur A renvoie $G = G' \cup \{a_1\}$ de même cardinal que I , donc optimal. ■

IV Principe général des preuves d'optimalité

L'idée de la preuve précédente et de celle qui vont suivre est d'identifier dans la solution gloutonne ce qui correspond au choix glouton initial. On va reprendre de manière générale le principe de cette preuve ici dans le cas d'un problème de minimisation.

Si on note g le glouton pour l'entrée E et c ce choix, on peut déduire de E la sous-entrée de E' correspondant à avoir enlevé c , ce qui peut conduire à changer E ou à supprimer d'autres éléments conséquences du choix c . Comme le glouton est une itération de choix gloutons, on déduit g' de g en omettant ce choix c et g' est le résultat du glouton pour E' .

Pour Huffman, E' se déduit en rajoutant une nouvelle lettre, ce n'est donc pas juste E sans les deux lettres les moins fréquentes.

On va démontrer un premier résultat qui est un résultat d'échange : si le glouton fait le choix c c'est que c est localement minimal, on va alors montrer qu'il existe un optimal o' faisant le choix c en montrant que si un optimal o a fait un choix minimal c' alors on peut préserver l'optimalité en remplaçant c' par c .

Dans le cas de Huffman, il s'agissait de placer les deux lettres les moins fréquentes en tant que sœurs d'un nœud profond.

Ensuite, on montre un résultat de réduction en enlevant c qui va relier $\chi(g)$ à $\chi(g')$ et, de même, $\chi(o')$ à $\chi(o'')$, car on a effectué la même omission de c pour le déduire. On s'assure que cette relation permette alors de faire le raisonnement suivant :

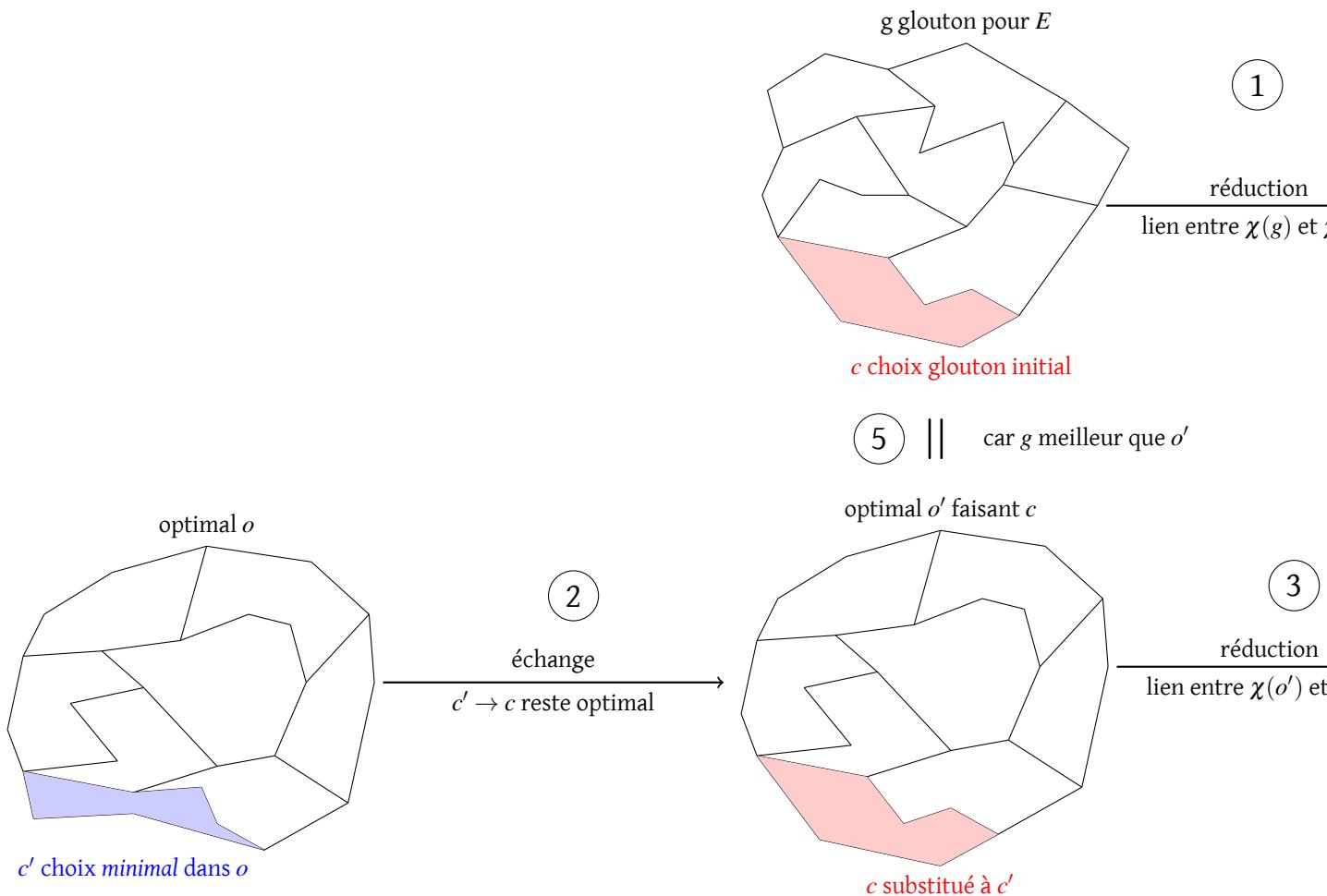
$$\chi(g) = \text{expr. de } \chi(g') \text{ et } c$$

Or, $\chi(g') = Opt(E') \leq \chi(o')$ donc

$$\chi(g) \leq \text{expr. de } \chi(o') \text{ et } c = \chi(o) = Opt(E)$$

Et par minimalité de $Opt(E)$, on a donc $\chi(g) = Opt(E)$.

Ce principe est décrit sur le schéma suivant :



V Ordonnancement de tâches

V.1 Description

On considère ici un problème voisin du problème précédent. On considère n tâches $T = \{t_1, \dots, t_n\}$ prenant une unité de temps pour être traitées sur une unité de calcul.

Chaque tâche t dispose d'une date limite $f(t) \in \llbracket 1, n \rrbracket$ (**deadline**) à laquelle elle doit être traitée sans quoi on écope d'une pénalité $p(t) \in \mathbb{N}$.

On appelle stratégie d'ordonnancement une application $d : T \rightarrow \llbracket 0, n - 1 \rrbracket$ qui associe à chaque tâche un unique temps de début $d(t)$. Selon cette stratégie, on déduit une séparation de T en deux ensembles disjoints :

- $T^+(d)$ l'ensemble des tâches traitées dans les délais : $t \in T^+(d) \iff d(t) < f(t)$.
- $T^-(d)$ l'ensemble des tâches traitées en retard : $t \in T^-(d) \iff d(t) \geq f(t)$.

On note alors $P(d) = \sum_{t \in T^-(d)} p(t)$ la somme des pénalités des tâches en retard.

■ **Exemple 15.4** On considère l'ensemble de tâches :

t_i	1	2	3	4	5	6	7
$f(t_i)$	1	2	3	4	4	4	6
$p(t_i)$	3	6	4	2	5	7	1

Une stratégie d'ordonnancement (les tâches en retard sont en gras) est donnée dans le tableau suivant :

t_i	1	2	3	4	5	6	7
$d(t_i)$	6	0	1	4	3	2	5

On a alors $P(d) = 5$.

On cherche à obtenir une stratégie d'ordonnancement de valeur $P(d)$ minimale.

On remarque que l'ordonnancement des tâches en retard n'a aucune importance, et on peut donc se contenter de déterminer une stratégie d'ordonnancement pour les tâches traitées dans les délais et la compléter par n'importe quel ordonnancement des autres tâches. On peut ainsi reformuler le problème : déterminer un sous-ensemble $T^+ \subset T$ de tâches **pouvant** être traitées dans les délais tel que $\sum_{t \in T^+} p(t)$ soit **maximale**.

V.2 Algorithme glouton et implémentation

On résout maintenant ce problème de maximisation des pénalités T^+ par un algorithme glouton :

- On commence avec $T^+ = \emptyset$ et tous les temps de $\llbracket 0, n - 1 \rrbracket$ sont marqués comme étant disponibles.
- On parcourt les tâches dans l'ordre décroissant des pénalités.
 - ★ Quand on considère la tâche t s'il existe un temps i disponible tel que $i < d(t)$ alors on marque comme indisponible le temps $i_0 = \max \{ i \in \llbracket 0, n - 1 \rrbracket \mid i < d(t) \text{ et } i \text{ disponible} \}$ et on rajoute alors t à T^+ en commençant t au temps i_0 .
- On place les tâches restantes aux temps disponibles.

Pour les structures de données, on utilise une représentation en tableaux de booléens (des `unsigned char` à 0 ou 1 en C) pour la disponibilités des temps. L'ensemble T^+ est alors implicite car il correspond aux tâches ordonnancé dans la première étape. Utiliser un tableau implique qu'une recherche linéaire soit faite pour chercher un plus grand temps disponible, et donc, la complexité temporelle globale sera en $O(n^2)$.

■ **Remarque 15.7** Il est possible d'améliorer cela pour passer en $O(n \log_2 n)$ (exercice).

Le programme C suivant implémente cet algorithme.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    unsigned int id;
    unsigned int date_limite;
    unsigned int penalite;
    int debut; /* -1 tant que la tâche n'est pas ordonnancée */
} tache;

int compare_taches(const void *t1, const void *t2)
{
    return ((tache *)t2)->penalite - ((tache *)t1)->penalite;
}

void *ordonnancement(tache *taches, size_t nb_taches)
{
    unsigned char *temps_occupe = malloc(sizeof(unsigned char) * nb_taches);
    memset(temps_occupe, 0, nb_taches);

    /* tri des activités par ordre décroissant des pénalités */
    qsort(taches, nb_taches, sizeof(tache), compare_taches);

    /* T+ par algorithme glouton */
    for (size_t k = 0; k < nb_taches; k++)
    {
        int i0 = -1;
        for (size_t i = 0; i < nb_taches; i++)
        {
            if (temps_occupe[i] == 0 && i < taches[k].date_limite)
                i0 = i;
        }
        if (i0 >= 0)
        {
            taches[k].debut = i0;
            temps_occupe[i0] = 1;
        }
    }

    /* Complétion par les tâches en retard */
    int i = 0; // indice du dernier temps disponible utilisé

    for (size_t k = 0; k < nb_taches; k++)
    {
        if (taches[k].debut == -1)
        {
            while(temps_occupe[i] == 1)
                i++;
            taches[k].debut = i;
            temps_occupe[i] = 1;
        }
    }
}
```

```

        free(temps_occupe);
    }

int main()
{
    tache taches[] = {
        { 1, 1, 3, -1 }, { 2, 2, 6, -1 }, { 3, 3, 4, -1 },
        { 4, 4, 2, -1 }, { 5, 4, 5, -1 }, { 6, 4, 7, -1 },
        { 7, 6, 1, -1 }
    };

    size_t nb_taches = sizeof(taches) / sizeof(tache);

    ordonnancement(taches, nb_taches);

    for (size_t i = 0; i < nb_taches; i++)
    {
        printf("T%d (f:%d,p:%d) @ %d\n", taches[i].id, taches[i].date_limite,
               taches[i].penalite, taches[i].debut);
    }

    return 0;
}

```

Il produit la sortie :

```

T6 (f:4,p:7) @ 3
T2 (f:2,p:6) @ 1
T5 (f:4,p:5) @ 2
T3 (f:3,p:4) @ 0
T1 (f:1,p:3) @ 4
T4 (f:4,p:2) @ 6
T7 (f:6,p:1) @ 5

```

Ce qui correspond à l'ordonnancement $t_3, t_2, t_5, t_6, t_1, t_7, t_4$. Les tâches t_1 et t_4 sont en retard, donc la pénalité totale est de 5.

V.3 Preuve d'optimalité

On va montrer que cet algorithme glouton renvoie un ensemble T^+ optimal. Pour cela, on procède comme précédemment. Tout d'abord, on montre qu'il existe une solution optimale qui effectue le premier choix de l'algorithme glouton.

Théorème V.1 Soit T un ensemble de tâches et $t \in T$ une tâche de pénalité maximale. Il existe un ensemble T^+ de tâches pouvant être traitées dans les délais, maximal pour les pénalités et tel que $t \in T^+$.

Démonstration.

Soit $T^+ \subset T$ un ensemble maximal. S'il contient t , il convient directement. Sinon, il existe une tâche t' de T^+ qui est traitée à un moment où on pourrait traiter t à temps (sinon $T^+ \cup \{t\}$ conviendrait et T^+ ne pourrait être maximal). On a $p(t') \leq p(t)$ par maximalité de t . L'ensemble T' déduit de T^+ en remplaçant t' par t convient car on a forcément $P(T') = P(T^+)$ (en fait \geq mais = par optimalité de T^+) et par construction toutes ses tâches peuvent être traitées à temps. ■

On montre maintenant qu'en enlevant le choix glouton, on obtient une solution optimale du sous-problème.

Théorème V.2 Soit $T^+ \subset T$ ensemble de tâches pouvant être traitées, maximal pour les pénalités et contenant une tâche t de plus grande pénalité. Soit i l'instant auquel la tâche t commence dans un ordonnancement de T^+ .

On pose $T' = T \setminus \{t\}$ avec des dates limites modifiées :

$$\forall t' \in T', d_{T'}(t') = \begin{cases} d_T(t') & \text{si } d_T(t') \leq i \\ d_T(t') - 1 & \text{sinon} \end{cases}$$

$T^+ \setminus \{t\}$ est alors maximal pour T' .

Démonstration.

Dans T' , on a à la fois enlevé t et supprimé l'instant i . Tout ordonnancement de T' peut alors être relevé en un ordonnancement de T en décalant d'un instant les tâches commençant à partir de l'instant i et en ordonnant là la tâche t . Réciproquement d'un ordonnancement dans T , on déduit directement un ordonnancement de T' .

Ainsi, s'il existait T'^+ maximal pour T' tel que $P(T'^+) > P(T^+ \setminus \{t\}) = P(T^+) - p(t)$ alors $T'^+ \cup \{t\}$ serait de somme de pénalités strictement plus grande que celle de T^+ supposé maximal.

Donc, $T^+ \setminus \{t\}$ est maximal. ■

On conclut alors directement par récurrence sur le nombre de tâches comme on l'a fait précédemment pour la sélection d'activités :

Théorème V.3 L'algorithme glouton renvoie un ordonnancement optimal.

VI Exercices

Beaucoup de ces exercices sont des adaptations d'exercices du livre Algorithms de Jeff Erickson.

Exercice 15.2 Un bibliothécaire souhaite ranger des collections de livres classées par auteur sur une longue étagère. On ainsi une suite (a_1, \dots, a_n) de collections données par la taille de la collection sur l'étagère, par exemple, en nombre de pages.

Un rangement des livres consiste à ordonner les collections de la première à la dernière sur l'étagère. Plus formellement, il s'agit d'une permutation $\sigma \in \mathfrak{S}_n$ où $\sigma(i)$ donne le numéro de la collection numéro i dans l'étagère.

Pour accéder à un auteur, il est nécessaire de parcourir linéairement l'étagère en partant de la première collection. Le coût d'accès à la k -ième collection est donc $\text{cout}(k) = \sum_{i=1}^k a_{\sigma(i)}$.

Le coût moyen d'accès aux collections est alors

$$\frac{1}{n} \sum_{k=1}^n \text{cout}(k)$$

Déterminer un algorithme permettant d'obtenir un rangement de coût minimal et l'ana-

lyser (correction, complexité).

Exercice 15.3 Soit X un ensemble de n segments de \mathbb{R} . On dit que $Y \subset X$ couvre X si $\bigcup_{y \in Y} y = \bigcup_{x \in X} x$. Le cardinal de Y est appelé la taille de la couverture. On cherche ici une couverture de taille minimale.

- Déterminer un algorithme glouton pour obtenir une couverture *a priori* de petite taille.
- Renvoie-t-il une couverture de taille minimale ?

Démonstration.

- On note $I_1 = [a_1; b_1], \dots, I_n = [a_n; b_n]$ les segments. Pour réaliser un algorithme glouton, on va considérer le plus petit élément x de $E = \bigcup I_i$. Comme ce sont des segments et qu'il y en a un nombre fini, ce plus petit élément existe toujours. Or, x doit être couvert et pour cela, on va considérer le plus grand des segments le contenant. Comme c'est le minimum, il est forcément de la forme $I = [x; y]$. On pose alors $I'_i = \overline{I_i \setminus I}$ où

$$\emptyset = \emptyset \quad \overline{[a; b]} = \overline{[a; b]} = \overline{[a; b]} = [a; b]$$

Comme x ne peut être intérieur à un des I_i , la différence $I_i \setminus I$ est nécessairement un intervalle, donc d'une de ces quatre formes (le cas $]a; b[$ étant exclu car on retranche un segment à un segment). On a $E = I \cup \bigcup I'_i$ et on déduit une couverture de $\bigcup I'_i = I'_{i_1} \cup \dots \cup I'_{i_p}$ par l'algorithme glouton (on omet alors les ensembles vides qui ne sont pas des segments mais ne contribuent pas à la couverture ou à E). Comme $I'_i \subset I_i$ on en déduit une couverture pour E .

- Comme pour les autres preuves d'optimalité, on va la décomposer en deux temps.
 - Tout d'abord, il existe une couverture optimale réalisant le choix glouton, c'est direct car une couverture optimale doit couvrir le minimum x et elle contient ainsi un segment de la forme $I' = [x; y'] \subset [x; y] = I$ le plus grand segment. Donc on peut substituer I à I' tout en préservant la minimalité de la couverture.
 - Ensuite, on raisonne par récurrence forte sur le nombre de segments pour montrer que le glouton est optimal. S'il n'y a qu'un segment, c'est direct. Si la propriété est vraie pour $1 \leq p < n$ segments et qu'on considère n segments I_1, \dots, I_n , alors on peut supposer que le choix glouton est I_1 quitte à réordonner les segments. Il existe donc un optimal $I_1 \cup I_{i_1} \cup \dots \cup I_{i_r} = E$ et le glouton donne lui $I_1 \cup I_{j_1} \cup \dots \cup I_{j_s} = E$ avec $1 + s \geq 1 + r$. On pose $I'_{j_k} = \overline{I_{j_k} \setminus I_1}$ et on a donc, par construction, $E \subset \bigcup I'_{j_k}$ et $E \subset I \cup \bigcup I'_{i_k}$. Or, la couverture $\bigcup I'_{j_k}$ est optimale par hypothèse de récurrence (on a au moins enlevé $I'_1 = \emptyset$) donc $s \leq r$. On en déduit directement que $s = r$.

■ **Note 15.1** La preuve peut être simplifiée tout en préservant son intérêt en ne considérant que des segments d'entiers $\llbracket a, b \rrbracket$. Ici, le problème vient du fait que si on a les segments $[1; 3]$ et $[2; 4]$, une fois $[1; 3]$ choisi, il faut couvrir $[2; 4] \setminus [1; 3] = [3; 4]$ ce qui oblige à considérer $[3; 4]$.

Exercice 15.4 On considère le problème du rendu de monnaie présenté dans le premier paragraphe.

- Montrer que, pour le système (b^0, b^1, \dots, b^k) où $b \in \mathbb{N}, b \geq 2$ et $k \in \mathbb{N}$, l'algorithme glouton est optimal.

- b) Déterminer un algorithme renvoyant le nombre de pièces optimal pour un système monétaire quelconque. *Indice : ce n'est certainement pas l'algorithme glouton.*

Démonstration.

- a) Commençons par montrer que si $x \in \mathbb{N}^*$ tel que b^i soit la plus grande pièce telle que $b^i \leq x$, alors il existe un rendu de monnaie optimal utilisant une pièce b^i . En effet, soit (r_0, r_1, \dots, r_k) un rendu optimal. S'il n'utilise pas la pièce b^i , alors il utilise uniquement les pièces b^0 à b^{i-1} . Comme $(b-1) \sum_{k=0}^{i-1} b^i = b^i - 1 < b^i \leq x$ il y a forcément une pièce b^j avec $r_j \geq b$. Comme $r_j b^j = (r_j - b)b^j + b^{j+1}$, on peut réduire le rendu de $b-1 \geq 1$ pièces ainsi. Comme il ne serait pas optimal, l'optimal utilise nécessairement la pièce b^i . On déduit alors par récurrence rapide que le glouton renvoie l'optimal. Ici, le fait qu'on soit forcé d'utiliser la pièce b^i montre en fait qu'il n'y a qu'un unique rendu optimal. Celui-ci est, en vérité, l'écriture de x dans la base (b^0, b^1, \dots, b^k) pour les nombres $< b^{k+1}$.
- b) On remarque que $Opt(x)$, le nombre de pièces optimal pour x , vérifie, pour $x > 0$, $Opt(x) = 1 + \min_{p_i \leq x} Opt(x-p_i)$. On en déduit un algorithme *a priori* naïf consistant à résoudre cette récurrence en identifiant la pièce p_i réalisant le minimum à chaque appel. On verra dans le chapitre sur la programmation dynamique qu'il est possible de rendre cet algorithme efficace en calculant ces minimums dans le bon ordre ou se souvenant des résultats des appels.

Exercice 15.5 On considère un tableau $t = [a_0, \dots, a_{n-1}]$ de n entiers relatifs. Un couple (i, j) est appelé un **segment positif pour t** si $1 \leq i \leq j \leq n$ et $\sum_{k=i}^j a_k \geq 0$.

On dit qu'une suite de segments positifs couvre t si chaque élément positif de t a son indice dans un des segments.

Si on considère le tableau :

$$t = [| 3; -5; 7; -4; 1; -8; 3; -7; 5; -9; 5; -2; 4 |]$$

Alors $(0, 4)$ (de somme $3 - 5 + 7 - 4 + 1 = 2$), $(6, 8)$ (de somme $3 - 7 + 5 = 1$) et $(10, 12)$ (de somme $5 - 2 + 4 = 7$) est une suite de 3 segments couvrant t .

Si t ne contient que des valeurs négatives, on considère qu'il est couvert par la suite vide 0 segment.

Déterminer un algorithme permettant d'obtenir une suite couvrant t de plus petit cardinal et analyser celui-ci.

Exercice 15.6 On considère le processus suivant : on part de l'entier 1 et à chaque étape on peut soit doubler l'entier courant, soit lui ajouter un. L'objectif est d'atteindre un entier cible n .

Par exemple, on peut atteindre 10 en quatre étapes ainsi :

$$1 \xrightarrow{+1} 2 \xrightarrow{\times 2} 4 \xrightarrow{+1} 5 \xrightarrow{\times 2} 10$$

Tout entier peut être atteint par une succession d'incrément, mais ce n'est pas le plus court en terme de nombre d'étapes.

Déterminer un algorithme permettant d'obtenir le nombre d'étapes minimal pour at-

teindre un entier n et l'analyser.

Démonstration.

On va considérer la stratégie suivante pour obtenir 1 en un minimum d'étapes en partant de n :

- Si n est pair, alors on le divise par 2
- Si n est impair, on lui retranche 1.

Notons $G(n)$ le nombre d'étapes pour cette stratégie gloutonne. On a par construction $G(1) = 0$ et pour $p \geq 1$, $G(2p) = 1 + G(p)$ et $G(2p+1) = 1 + G(2p) = 2 + G(p)$.

Le plus simple pour montrer que cette stratégie est optimale est de considérer le même type de formule pour l'optimal. En effet, $Opt(1) = 0$ et pour $p \geq 1$, $Opt(2p+1) = 1 + Opt(2p)$ et $Opt(2p) = 1 + \min(Opt(p), Opt(2p-1))$. On en déduit ainsi par une récurrence immédiate que $\forall n \geq 2$, $Opt(n) \leq 1 + Opt(n-1)$.

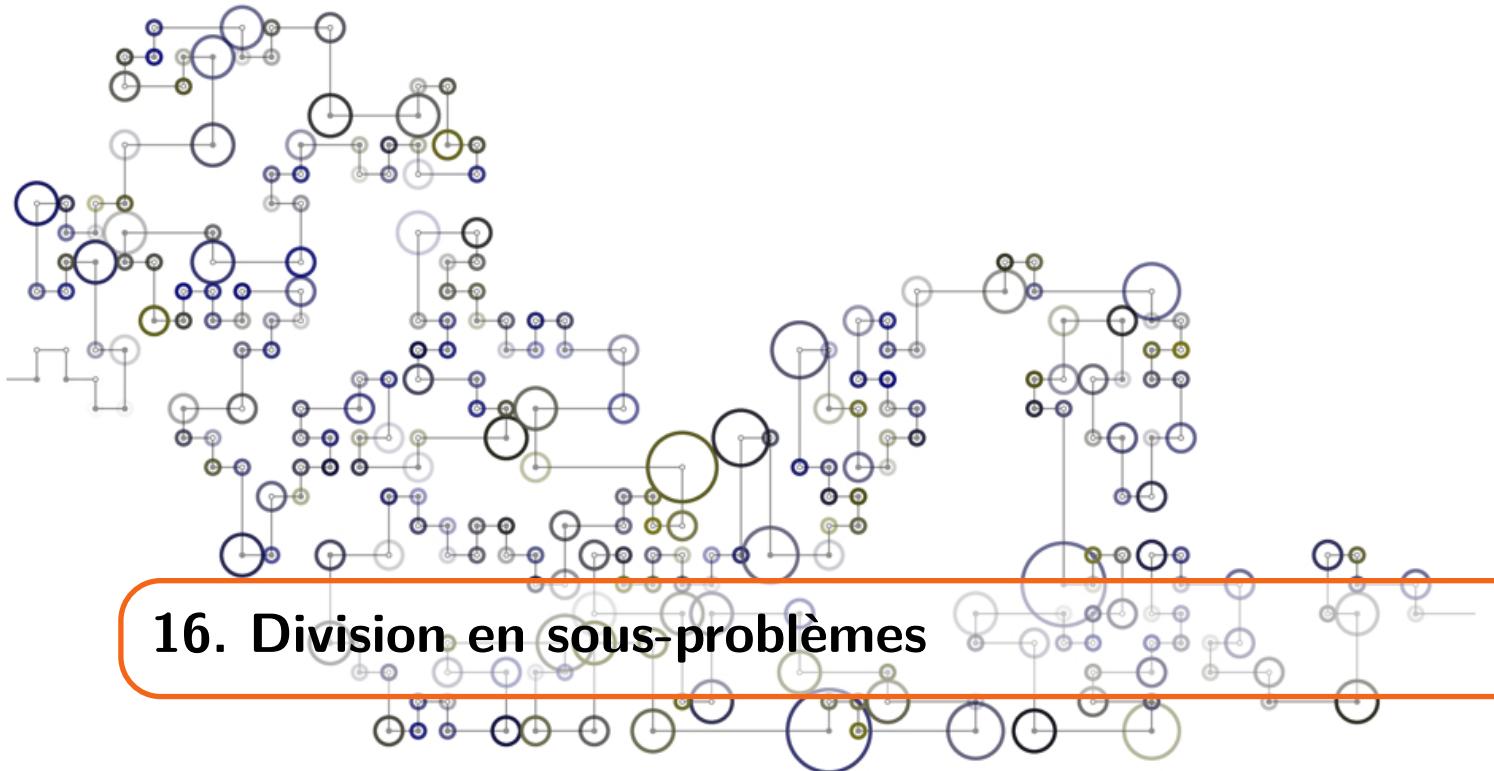
On va montrer par récurrence **forte** que $\forall n \geq 1$, $Opt(n) = G(n)$.

- Initialisation : on a vu que $Opt(1) = G(1) = 0$.
- Hérédité : Soit $n > 1$ tel que la propriété soit vérifiée pour tout entier $1 \leq p < n$. Par disjonction de cas selon que n soit pair ou non :
 - * si $n = 2p+1$, alors $G(2p+1) = 1 + G(2p) = 1 + Opt(2p) = Opt(2p+1)$
 - * si $n = 2p$, quitte à traiter à part le cas trivial $n = 2$, on peut supposer que $p > 1$. On a $G(2p) = 1 + G(p)$ et $Opt(2p) = 1 + \min(Opt(p), Opt(2p-1))$. Or

$$\begin{aligned} Opt(2p-1) &= G(2p-1) = 1 + G(2p-2) = 2 + G(p-1) \\ &= 2 + Opt(p-1) > 1 + Opt(p-1) \geq Opt(p) \end{aligned}$$

Donc $Opt(2p) = 1 + Opt(p) = 1 + G(p) = G(2p)$.

Dans tous les cas, on a bien la propriété voulue.



16. Division en sous-problèmes

Dans ce chapitre, on étudie des algorithmes ayant la particularité de reposer sur une division d'un problème en plusieurs sous-problèmes :

- soit pour les résoudre récursivement, on parle alors de la méthode *diviser pour régner*
- soit pour résoudre ces sous-problèmes, peut-être d'une manière différente, afin d'obtenir une solution plus efficace au problème principal, on parle de *rencontre au milieu* (ou *meet in the middle*)

I Diviser pour régner

1.1 Principe

Le principe des algorithmes basé *Diviser pour régner* est de décomposer un problème en plusieurs sous-problèmes disjoints et de déduire des solutions de ces sous-problèmes une solution au problème de départ.

Le point clé pour ce principe est de pouvoir **fusionner** les solutions de sous-problèmes pour en faire une solution, et de pouvoir le faire dans un temps/espace raisonnable. On procède alors par récursivité en appliquant ce principe pour résoudre les sous-problèmes eux-mêmes jusqu'à tomber sur des sous-problèmes très simples.

Ainsi, si on a une entrée E , on va en déduire des entrées E_1, \dots, E_k pour des sous-problèmes tels que $|E_i| \leq \left\lceil \frac{|E|}{p} \right\rceil$, où p et k sont indépendants de E . Une fois les solutions S_1, \dots, S_k à ces sous-problèmes obtenus par des appels récursifs, on va en déduire par une **fusion** des solutions, la solution S au problème initial.

En terme de complexité temporelle, si $T(n)$ est la complexité de résolution pour une solution de taille n , on aura :

$$T(n) \leq kT\left(\left\lceil \frac{n}{p} \right\rceil\right) + f(n)$$

où $f(n)$ est le coût de la fusion et en supposant, ce qui est très raisonnable, que T est croissante.

■ **Remarque 16.1** La croissance de T est naturellement déduite du fait que le problème sur une entrée de taille n est **plus facile** que le problème sur une entrée de taille $p \geq n$. On peut ainsi penser au tri d'un tableau où il suffit de rajouter des éléments factices au début ou à la fin pour se ramener à un problème sur une entrée de taille plus grande.

On ne donnera pas ici de théorème général permettant d'exprimer $T(n)$ selon les différentes valeurs de $f(n)$, on se contentera de faire des preuves dans des cas particuliers.

1.2 Somme d'éléments dans un tableau

On considère un tableau t de n nombres dont on cherche à calculer la somme $S(t) = \sum_{i=0}^{n-1} t[i]$. Il est possible de faire ce calcul très aisément avec une simple boucle :

```
C | int somme(int *t, size_t nb)
{   int s = 0;
    for(int i = 0; i < nb; i++)
    {
        s = s + t[i];
    }
    return s;
}
```

Il est possible, bien que cela ne soit pas naturel, de traiter ce problème avec un algorithme *diviser pour régner*. En effet, on peut couper le tableau en deux, calculer les deux sous-sommes S_1 et S_2 , puis les fusionner de manière triviale en calculant la somme $S = S_1 + S_2$.

```
Ocaml | let rec somme_div t i j =
|     if i = j
|     then t.(i)
|     else
|         let m = i + (j-i)/2 in
|         let s1 = somme_div t i m in
|         let s2 = somme_div t (m+1) j in
|             s1 + s2
|
| let somme t = somme_div t 0 (Array.length t - 1)
```

Si $T(n)$ est le coût d'une somme d'un tableau à n éléments, on a donc $T(1) = 1$ et

$$\forall n \geq 2, T(n) \leq 2T(\lceil n/2 \rceil) + O(1)$$

■ **Remarque 16.2** On constate ici qu'on utilise $\lceil x \rceil$ la partie entière supérieure liée à une majoration. En effet, si on coupe l'entrée en deux et qu'elle est de taille impaire $2p + 1$, une des deux moitiés sera de longueur $p + 1$ qui est la partie entière supérieure.

On aura alors $T(n) = T(p) + T(p + 1) + O(1) \leq 2T(p + 1) + O(1)$ par croissance de T .

Les résolutions de récurrence vont se faire en deux temps : un cas où toutes les divisions tombent justes, c'est-à-dire ici une puissance de deux, et un théorème où on s'y ramène pour un entier quelconque par encadrement et croissance de T .

Lemme I.1 $\forall n \in \mathbb{N}, T(2^n) = O(2^n)$

Démonstration.

On commence par expliciter le $O(1)$ présent dans la récurrence : il existe $M \geq 1$ tel que $\forall n \in \mathbb{N}^*, T(n) \leq 2T(\lceil n/2 \rceil) + M$.

Montrons par récurrence que $\forall n \in \mathbb{N}, T(2^n) \leq M2^n$

a) On a $T(2^0) = 1 \leq M2^0$.

b) Si $T(2^n) \leq M2^n$ alors $T(2^{n+1}) \leq 2T(2^n) + M \leq M2^n + M \leq M2^{n+1}$.

On a ainsi $T(2^n) = O(2^n)$. ■

Théorème I.2 $\forall n \geq 1, T(n) = O(n)$.

Démonstration.

Soit $n \in \mathbb{N}^*$, il existe $p \in \mathbb{N}$ tel que $2^p \leq n < 2^{p+1}$. Par croissance de T , on a $T(n) \leq T(2^{p+1})$. Or, il existe M' tel que $\forall k, T(2^k) \leq M'2^k$. $T(n) \leq M'2^{p+1} \leq 2M'n$.

Donc, $T(n) = O(n)$. ■

I.3 Tri fusion

L'algorithme du tri fusion est un des exemples les plus importants d'algorithmes *Diviser pour régner* :

- Étant donnée une liste l de taille $n \geq 2$, on va considérer les sous-listes l_p des valeurs d'indice pair et l_i des valeurs d'indice impair.
- On trie ensuite l_1 et l_2 pour obtenir l'_1 et l'_2 .
- On fusionne ces deux listes pour obtenir $l' = \text{fusion}(l'_1, l'_2)$ liste triée déduite de l .

Comme expliqué dans le paragraphe précédent, les tris de l_1 et l_2 s'effectuent eux-aussi à l'aide d'un tri fusion.

■ **Note 16.1** TODO : dessin ■

Voici une implémentation en OCaml de cet algorithme :

```
let rec separe_en_deux l =
  match l with
  | [] -> ([], [])
  | [x] -> ([x], [])
  | x::y::q -> let l1, l2 = separe_en_deux q in
    (x::l1, y::l2)

let rec fusionne l1 l2 =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | x::q1, y::q2 ->
    if x < y
    then x :: (fusionne q1 l2)
    else y :: (fusionne l1 q2)

let rec tri_fusion l =
```

```

match l with
| [] -> []
| [x] -> [x]
| _ ->
    let l1, l2 = separe_en_deux l in
    let l1p = tri_fusion l1 in
    let l2p = tri_fusion l2 in
    fusionne l1p l2p

```

La correction et la terminaison de cet algorithme ne posant aucune difficulté, on va se concentrer sur le calcul de la complexité temporelle :

- `separe_en_deux` consiste en un parcours linéaire de la liste `l` donc $O(|l|)$.
- `fusionne` supprime un élément d'une des deux listes à chaque appel récursif, donc une complexité en $O(|l_1| + |l_2|)$.
- Pour `tri_fusion` la situation est plus complexe en raison du double appel récursif. On va d'abord traiter le cas des listes contenant 2^k éléments.

Notons t_n la complexité temporelle pour $|l| = n$.

Lemme I.3 $t_{2^n} = O(2^n \log_2 2^n)$

Démonstration.

Par l'analyse de complexité des deux fonctions auxiliaires, on a pour $n \in \mathbb{N}$

$$t_{2^{n+1}} = 2t_{2^n} + O(2^n) \leq 2t_{2^n} + M2^n$$

où on peut supposer que $M \geq 1$.

On va montrer par récurrence sur $n \in \mathbb{N}^*$ que $t_{2^n} \leq 2Mn2^n$.

- Initialisation : $t_{2^1} = 2t_1 + M2 = 2M + 2 \leq 4M = 2 \times 1 \times 2^1 M$.
- Hérédité : si $n \in \mathbb{N}^*$ et l'hypothèse est vérifiée pour t_{2^n} , alors $t_{2^{n+1}} \leq 4n2^n M + M2^n = (4n + 1)M2^n \leq 4(n + 1)M2^n \leq 2M(n + 1)2^{n+1}$.

Ainsi $t_{2^n} = O(n2^n)$. ■

Théorème I.4 $t_n = O(n \log_2 n)$

Démonstration.

Le lemme assure qu'il existe M' tel que $\forall p \in \mathbb{N}^*, t_{2^p} \leq M'p2^p$.

Soit $n \in \mathbb{N}^*$ on sait qu'il existe p tel que $2^{p-1} \leq n < 2^p$ et donc $p - 1 \leq \log_2 n < p$. Par croissance de t (direct car on a plus d'éléments à trier) on a $t_n \leq t_{2^p} \leq M'p2^p \leq 2M'(1 + \log_2 n)n = O(n \log_2 n)$.

Ainsi, $t_n = O(n \log_2 n)$. ■

■ **Remarque 16.3** On a utilisé implicitement la croissance de t_n ici : plus la liste est longue, plus on effectue d'opérations. ■

Le programme suivant présente une implémentation du tri fusion reposant sur des tableaux. Les sous-tableaux sont manipulés à l'aide de leurs indices de début et de fin comme pour la recherche dichotomique.

OCaml

```

let rec separe_en_deux l =
  match l with
  | [] -> ([], [])
  | [x] -> ([x], [])
  | x::y::q -> let l1, l2 = separe_en_deux q in
    (x::l1, y::l2)

let rec fusionne l1 l2 =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | x::q1, y::q2 ->
    if x < y
    then x :: (fusionne q1 l2)
    else y :: (fusionne l1 q2)

let rec tri_fusion l =
  match l with
  | [] -> []
  | [x] -> [x]
  | _ ->
    let l1, l2 = separe_en_deux l in
    let l1p = tri_fusion l1 in
    let l2p = tri_fusion l2 in
    fusionne l1p l2p

```

■ **Note 16.2** TODO : exercice tri avec un tableau et tri **en place**

I.4 Recherche dichotomique

On a déjà vu la recherche dichotomique, ici, on coupe le tableau en deux mais on n'effectue qu'un seul appel récursif, la récurrence est donc en

$$T(n) = T(\lceil n/2 \rceil) + O(1)$$

et on obtient facilement $T(n) = O(\log_2 n)$.

I.5 Principe général d'analyse des récurrences

On remarque ici qu'on retombe souvent sur la même méthode pour analyser des récurrences de la forme

$$\forall n \in \mathbb{N}^*, T(n) \leq kT\left(\left\lceil \frac{n}{p} \right\rceil\right) + f(n)$$

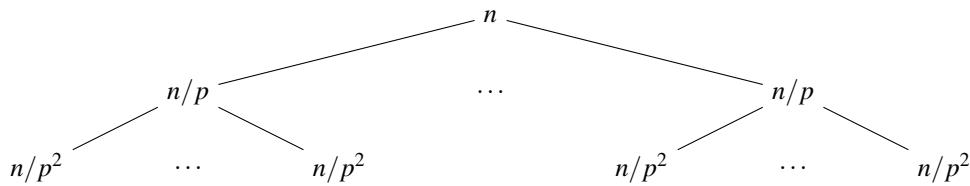
Comme on ne considère que des suites positives et croissantes, on peut se ramener à une égalité par majoration. En effet, si S vérifie $S(0) = T(0)$ et $\forall n \in \mathbb{N}^*, S(n) = kS\left(\left\lceil \frac{n}{p} \right\rceil\right) + f(n)$ alors on a $\forall n \in \mathbb{N}, T(n) \leq S(n)$ par une récurrence immédiate et d'une majoration de $S(n)$ on en déduira directement une majoration pour $T(n)$.

Cette remarque peut donc se résumer ainsi :

■ **Remarque 16.4** Il est toujours possible de se ramener à une relation de récurrence avec une égalité.

Que signifie cette relation de récurrence : que pour résoudre l'instance de taille n on va faire k appels récursifs à des instances de taille au plus $\lceil n/p \rceil$ et le coup de la fusion des résultats sera $f(n)$.

On peut ainsi représenter les coûts de fusion cumulés qui sont associés à chaque niveau de l'arbre d'appels récursifs. Pour simplifier, on considère que $n = p^l$ ce qui permet de n'avoir que des divisions entières tout au long de l'arbre. On remarque qu'on peut s'y ramener par croissance en majorant n par une puissance de p et on qu'on a alors $l = O(\log_p n)$.



On a alors les coûts de fusion

$$T(n) = f(n) + kf(n/p) + k^2f(n/p^2) + \dots = \sum_{i=0}^l k^i f(n/p^i)$$

On a trois cas standard pour cette somme :

- Poids sur la racine, c'est $f(n)$ qui l'emporte sur les coûts et le reste de l'arbre n'a pas d'influence sur la complexité. Ici $T(n) = f(n)$.
- Poids sur les feuilles, le coût de fusion n'est pas important et c'est le nombre de feuilles qui compte, on a alors $T(n) = k^l f(1) = O(k^l) = O(k^{\log_p n}) = O(n^{\log_p k})$.
- Poids réparti uniformément, on a chaque $k^i f(n/p^i) = O(f(n))$ et donc $T(n) = O(f(n) \log n)$.

■ **Exemple 16.1**

- Dans le cas de la multiplication de Karatsuba (exercice) on a $T(n) = 3T(\lceil n/2 \rceil) + O(n)$ donc les niveaux sont de plus en plus peuplés, on est dans le second cas et le niveau final va l'emporter. Ainsi, $T(n) = O(n^{\log_2 3})$.
- Dans le cas du tri fusion $T(n) = 2T(\lceil n/2 \rceil) + O(n)$ on a un équilibre du travail total de fusion qui reste linéaire pour chaque niveau. Donc, on est dans le troisième cas et $T(n) = O(n \log n)$.

■ **Remarque 16.5** Pour retrouver des expressions rapidement, on peut étudier $u_n = \frac{T(p^n)}{k^n}$.

En effet, de la récurrence $T(p^n) = kT(p^{n-1}) + f(p^n)$ on déduit $u_n = u_{n-1} + \frac{f(p^n)}{k^n}$ et donc $u_n = \sum_{i=0}^n \frac{f(p^i)}{k^i} = O(g(n))$ qui se simplifie en général. Pour obtenir ensuite $T(p^n) = k^n u_n = O(k^n g(n))$ puis $T(n) = O(n^{\log_p k} g(\log_p n))$.

Par exemple, pour $T(n) = 2T(\lceil n/2 \rceil) + O(n)$ on a $u_n = T(2^n)/2^n = \sum_{i=0}^n \frac{O(2^i)}{2^i} = O(n)$ donc $T(2^n) = 2^n O(n) = O(n2^n)$ puis $T(n) = O(n \log n)$.

Autre exemple, pour $T(n) = 2T(\lceil n/2 \rceil) + O(1)$, on a $u_n = \sum_{i=0}^n 2^{-i} O(1) = O(1) \frac{1 - \frac{1}{2^{n+1}}}{1 - \frac{1}{2}} = O(1)$ car $\frac{1}{2^{n+1}} = o(1)$. Ainsi $T(2^n) = O(2^n)$ puis $T(n) = O(n)$.

1.6 Nombre d'inversions

Définition 1.1 Soit t une structure séquentielle (tableau, liste,) contenant des valeurs comparables a_0, \dots, a_{n-1} et énumérées dans cet ordre au sein de t .

Une paire $(i, j) \in \llbracket 0, n-1 \rrbracket^2$ où $i < j$ est appelée une *inversion* de t lorsque $a_i > a_j$.

On note $I(t)$ le nombre d'inversion de t .

- **Remarque 16.6**
 - Le nombre d'inversions permet de mesurer à quel point t est non triée dans l'ordre croissante.
 - Ce concept d'inversion est exactement celui utilisé pour les permutations en mathématiques : si $\sigma \in \mathfrak{S}_n$, il suffit de considérer $(\sigma(1), \dots, \sigma(n))$.

On cherche dans ce paragraphe à calculer $I(t)$ efficacement. Remarquons tout d'abord qu'un algorithme naïf est en $O(n^2)$ où $|t| = n$ en explorant toutes les paires :

```
size_t inversions(int *t, size_t taille)
{
    size_t inv = 0;
    for (size_t i = 0; i < taille; i++)
    {
        for (size_t j = i+1; j < taille; j++)
        {
            if (t[i] > t[j]) inv++;
        }
    }
    return inv;
}
```

On va maintenant donner un algorithme type *Diviser pour régner* :

- On sépare t en deux moitiés t_1 et t_2 .
- On calcule $I(t_1)$ et $I(t_2)$ par des appels récursifs.
- On compte les inversions entre des éléments de t_1 et des éléments de t_2
 - ★ Cela ne dépend pas de leur position dans t_1 ou dans t_2 .
 - ★ On peut donc trier t_1 en t'_1 et t_2 en t'_2 .
 - ★ On compte $N(t_1, t_2) = N(t'_1, t'_2)$ le nombre d'inversions entre t'_1 et t'_2 en $O(n)$ par l'algorithme ci-dessous.
- On en déduit que $I(t) = I(t_1) + I(t_2) + N(t_1, t_2)$.

- **Remarque 16.7** Pour calculer le nombre d'inversions entre deux tableaux triés t'_1 et t'_2 , on remarque que si $t'_1[i] > t'_2[j]$ alors $t'_1[i+1] \geq t'_1[i] > t'_2[j]$. Cela signifie qu'on peut compter les inversions de manière cumulative en avançant dans les deux tableaux en même temps :

OCaml

```

let inversions_croisees t1 t2 =
  let i = ref 0 in
  let j = ref 0 in
  let n1 = Array.length t1 in
  let n2 = Array.length t2 in
  let inv = ref 0 in
  let cumul = ref 0 in
  while !i < n1 && !j < n2 do
    if t1.(!i) > t2.(!j)
    then ( incr j; incr inv; incr cumul )
    else ( incr i; inv := !inv + !cumul )
  done;
  !inv + (!cumul * (n1 - !i - 1))

```

■ **Note 16.3** Rajouter un dessin illustrant l'aspect cumulatif.



Pour que cet algorithme fonctionne, il est nécessaire de trier à chaque étape les sous-tableaux, on obtient alors naïvement une récurrence en

$$T(n) \leq 2T(\lceil n/2 \rceil) + O(n \log n)$$

Qui se résout en $T(n) = O(n(\log n)^2)$. Il est possible de faire mieux en remarquant qu'on peut calculer le nombre d'inversions en décorant le tri fusion car celui fait naturellement calculer les sous-tableaux triés. On a donc une récurrence en

$$T(n) \leq 2T(\lceil n/2 \rceil) + O(n)$$

qui se résout en $T(n) = O(n \log n)$.

■ **Exercice 16.1** Écrire le programme réalisant cet algorithme.



Démonstration.

OCaml

```

let inversions_croisees t1 t2 =
  let i = ref 0 in
  let j = ref 0 in
  let n1 = Array.length t1 in
  let n2 = Array.length t2 in
  let inv = ref 0 in
  let cumul = ref 0 in
  while !i < n1 && !j < n2 do
    if t1.(!i) > t2.(!j)
    then ( incr j; incr inv; incr cumul )
    else ( incr i; inv := !inv + !cumul )
  done;
  !inv + (!cumul * (n1 - !i - 1))

let split t =
  let n = Array.length t in
  let m = n/2 in

```

```

Array.sub t 0 m, Array.sub t m (n-m)

let fusion t1 t2 =
  let n1 = Array.length t1 in
  let n2 = Array.length t2 in
  let t = Array.make (n1+n2) (t1.(0)) in
  let i = ref 0 in let j = ref 0 in
  for k = 0 to n1+n2-1 do
    if !j = n2 || (!i < n1 && t1.(!i) < t2.(!j))
    then ( t.(k) <- t1.(!i); incr i )
    else ( t.(k) <- t2.(!j); incr j )
  done;
  t

let rec inversions_div t =
  let n = Array.length t in
  if n = 1
  then t, 0
  else let t1, t2 = split t in
    let t1', n1 = inversions_div t1 in
    let t2', n2 = inversions_div t2 in
    let n12 = inversions_croisees t1' t2' in
    fusion t1' t2', n1+n2+n12

```

■

1.7 Points les plus proches

On considère le problème

Problème - PLUSPROCHEPAIRE

- Entrées :
Un ensemble P de n points dans le plan.
- Sortie : Une paire $\{p, p'\}$ de points telle que $dist(p, p') = \|\vec{pp'}\|$ soit minimale.

Ce problème a déjà été étudié dans le chapitre Recherche par force brute où l'algorithme naïf en $O(n^2)$ a été amélioré en un algorithme en $O(n \log n)$.

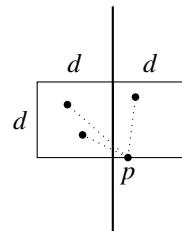
On présente ici un algorithme diviser pour régner.

Si $n > 1$ l'idée est de séparer les points en fonctions de la médiane des abscisses. C'est-à-dire de déterminer tel x , par exemple en prenant la moitié entre l'abscisse médiane et l'abscisse suivante, tel que $\lceil n/2 \rceil$ points soient d'abscisses $< x$, notons S_1 leur ensemble, et $\lfloor n/2 \rfloor$ soient d'abscisses $> x$, notons S_2 . Ensuite, on applique l'algorithme récursivement pour obtenir d_1 la plus petite distance dans S_1 et d_2 la plus petite distance dans S_2 . On pose $d = \min(d_1, d_2)$ et la question qui se pose est celle de calculer $d' = \min\{dist(p_1, p_2) \mid p_1 \in S_1, p_2 \in S_2\}$.

Comme on a déjà déterminé d , on peut se contenter de chercher à déterminer s'il existe $d' < d$. Pour cela, on peut se contenter de ne considérer dans S_1 que les points d'abscisse dans $]x - d; x]$ et pour S_2 dans $[x; x + d[$. On a ainsi une bande à étudier. Comme il n'est pas possible d'exclure qu'un nombre important de points soit dans cette bande, on a *a priori* une fusion en $O(n)$.

On va considérer les points de bas en haut dans la bande, par exemple avec un tri initial selon les ordonnées. Ainsi, il sera inutile, quand on considère un point p d'ordonnée y de considérer des points d'ordonnées $< y$. D'un autre côté, comme on cherche l'existence d'une distance $< d$, il est inutile de considérer des points d'ordonnée $> y + d$.

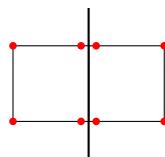
On se retrouve alors à comparer les points présents dans l'union de deux carrés de côté d de part et d'autres de la séparation :



Cependant, on sait que chaque carré étant situé d'un même côté de la ligne médiane de séparation, les points qui s'y trouvent sont à distance $\geq d$. Il est facile de se convaincre qu'il n'est pas possible de placer plus que quatre points dans un carré de côté de sorte qu'aucune paire soit à distance $< d$. Cette configuration correspond ainsi à placer les points aux sommets.

On en déduit donc qu'il y a maximum quatre points dans le carré gauche et quatre points dans le carré droit, dont le point p considéré : il n'est pas nécessaire d'explorer plus que les sept points suivants.

Sur le schéma suivant, on a volontairement séparé les points autour de la frontière pour marquer cette différence.



■ **Remarque 16.8** Cette analyse est très naïve, on peut démontrer qu'il suffit d'observer cinq autres points.

ERROR: `src/algorithme/.../.../snippets/algorithme/closest_div.c` does not exist

II Meet in the middle

II.1 Principe

II.2 Sous-ensemble de somme donnée

III Dichotomie pour passer de décision à optimisation

III.1 Principe

III.2 Couverture par des segments égaux

■ **Note 16.4** Il y a une confusion réel/entiers ici. À reprendre.

On considère ici n points sur la droite réelle. Le i -ème point est identifié par sa coordonnée x_i . On se pose alors, dans un premier temps, la question de savoir si on peut trouver k segments de longueur l tels que chaque point appartienne à au moins un de ces segments. Dans un second temps, on se posera la question de la longueur l minimale de ces segments.

III.2.i Couverture par des segments de longueur donnée

On va ici résoudre le premier problème :

Problème - EXISTENCECOUVERTURESEGMENT

- Entrées :
 - ★ n points sur la droite réelle x_1, \dots, x_n
 - ★ un entier $k \geq 1$
 - ★ un réel l
- Sortie : existe-t-il k segments S_1, \dots, S_k de longueur l tels que
 $\forall i \in \llbracket 1, n \rrbracket, \exists j \in \llbracket 1, k \rrbracket, x_i \in S_j$?

On remarque qu'un segment de longueur l est uniquement caractérisé par son extrémité gauche. Pour chaque x_i , il doit ainsi exister une extrémité gauche dans le segment $S_i = [x_i - l, x_i]$. On peut ainsi renverser le problème et en faire un problème de couverture de segments par des points : on cherche k points tel que chaque segment S_i contienne au moins un de ces points.

Problème - ENSEMBLEINTERSECTANTLIGNE

- Entrées :
 - ★ n segments $S_i = [l_i, r_i]$
 - ★ un entier $k \geq 1$.
- Sortie : Un ensemble P de k points tels que $\forall p \in P, \exists i \in \llbracket 1, n \rrbracket, p \in S_i$ en cas de succès

Ce problème peut se résoudre par un algorithme glouton :

- on commence avec $P = \emptyset$
- on trie les segments par r_i croissant
- pour chaque segment $S_i = [l_i, r_i]$, s'il ne contient aucun élément de P , on rajoute r_i à P .
- on répond avec un succès si $|P| \leq k$ (on peut alors compléter avec des points quelconques pour avoir exactement k points).

Cet algorithme est en $O(n \log n)$ en raison du tri initial.

III.2.ii Longueur minimale

On considère maintenant le problème suivant :

Problème - COUVERTURESEGMENTSMINIMALE

- Entrées :
 - ★ n points à coordonnées entières sur la droite réelle x_1, \dots, x_n
 - ★ un entier $k \geq 1$
- Sortie : la longueur l minimale pour laquelle il existe une couverture des points par k segments de longueur l .

On peut résoudre ce problème par dichotomie à l'aide de l'algorithme précédent

- on considère $L = \frac{\max_{1 \leq i < j \leq n} |x_j - x_i|}{k} = \frac{D}{k}$ où le diamètre D se calcule en $O(n)$ et correspond à répartir de manière uniforme les segments pour couvrir les points. Une telle couverture existe toujours.
- on peut considérer, sans avoir besoin de le calculer au préalable, le tableau V de booléen de longueur $L + 1$ tel que $V[l]$ indique s'il est possible de couvrir les points par k segments de longueur l . Pour obtenir la valeur $V[l]$ il suffit d'appliquer l'algorithme précédent.
- on effectue alors une recherche dichotomique du plus petit indice l tel que $V[l]$ soit vrai.

On obtient ainsi un algorithme en $O(n \log n \log \frac{D}{k})$ qu'on peut considérer comme étant en $O(n \log n)$ en supposant que D est une constante.

Cette dichotomie est en fait une instance d'un principe fondamental permettant de transformer un problème de décision (existe-t-il ?) en un problème d'optimisation (quelle est ... minimal/maximal?).

III.2.iii Implémentation

IV Problèmes

IV.1 Multiplication d'entiers

On considère ici le problème de la multiplication de deux entiers données en précision arbitraire par le tableau de leurs chiffres.

Ainsi, l'entier $123 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$ sera représenté par le tableau $[|3;2;1|]$ de telle sorte que le nombre représenté par t soit $\sum_{k=0}^{n-1} t[k]10^k$ où $n = |t|$ la taille de t . On suppose que t ne finit pas par des 0. Autrement dit : soit t est vide, soit la dernière valeur de t est non nulle.

On se convaincra aisément que les entiers naturels sont en bijection avec de telles représentations.

Question IV.1 Écrire une fonction `simpl : int array -> int array` qui renvoie un nouveau tableau obtenu en supprimant tous les 0 en fin de tableau : `simpl [|1;2;3;0;0|] = [|1;2;3|]`.

Question IV.2 Écrire une fonction `add : int array -> int array -> int array` qui calcule la somme de deux entiers en temps linéaire.

On va étudier ici plusieurs techniques de multiplication.

IV.1.i Multiplication naïve

On considère la multiplication posée comme étudiée à l'école primaire :

$$\begin{array}{r} 1 & 2 & 3 \\ \times & 4 & 2 \\ \hline 2 & 4 & 6 \\ 4 & 9 & 2 \\ \hline 5 & 1 & 6 & 6 \end{array}$$

Plus précisément, si $y = \sum_{j=0}^m y_j 10^j$, on a

$$x \times y = \sum_{j=0}^m (y_j \times x) 10^j$$

Question IV.3 Écrire une fonction `mult_digit : int -> int array -> int array` telle que `mult_digit a x` renvoie $a \times x$ où a est un chiffre et x un nombre.

Question IV.4 Écrire une fonction `mult_dec : int -> int array -> int array` telle que `mult_dec p x` renvoie $x \times 10^p$ où p est un entier et x un nombre.

Question IV.5 Écrire une fonction `mult_naive : int array -> int array -> int array` réalisant cette multiplication.

Question IV.6 Quelle est la complexité de cette méthode en fonction de la longueur des entrées ?

IV.1.ii Première approche diviser pour régner

On remarque que pour $0 \leq a, b, c, d < 10^m$:

$$(10^m a + b)(10^m c + d) = 10^{2m}ac + 10^m(ad + bc) + bd$$

On peut alors calculer récursivement les quatre produits ac, ad, bc et bd .

On en déduit donc une solution type **diviser pour régner** de la multiplication :

- $\text{mul}(x, y, 1) = x \times y$
- pour $n \geq 2$, $\text{mul}(x, y, n) = 10^{2m}\text{mul}(a, c, m) + 10^m(\text{mul}(a, d, m) + \text{mul}(b, c, m)) + \text{mul}(b, d, m)$ où $x = a10^m + b$, $y = c10^m + d$ et $m = \lceil n/2 \rceil$.
où $x, y \leq 10^n$.

Question IV.7 Déterminer avec précision la complexité $T(n)$ de $\text{mult}(x, y, n)$.

Que peut-on en conclure ?

IV.1.iii Algorithme de Karatsuba

Face à ce problème, Karatsuba, alors âgé de 23 ans, est parti de la remarque suivante :

$$ac + bd - (a - b)(c - d) = ad + bc$$

qui permet de voir qu'en calculant ac , bd et le produit $(a - b)(c - d)$ on peut en déduire le coefficient $ad + bc$ de 10^m , et ainsi économiser une multiplication.

On en déduit ainsi une nouvelle solution pour la multiplication :

- $\text{kar}(x, y, 1) = x \times y$
- pour $n \geq 2$, $\text{kar}(x, y, n) = 10^{2m}X + 10^m(X + Y - Z) + Y$ où où $x = a10^m + b$, $y = c10^m + d$, $m = \lceil n/2 \rceil$, $X = \text{kar}(a, c, m)$, $y = \text{kar}(b, d, m)$ et $Z = \text{kar}(a-b, c-d, m)$.

La présence de nombres négatifs ne complique pas la complexité car il suffit de rajouter un booléen indiquant le signe à côté du tableau des chiffres d'un nombre.

Question IV.8 Déterminer avec précision la complexité $K(n)$ de $\text{kar}(x, y, n)$.



17. Algorithmique des textes

Source image : justgrims, <https://www.flickr.com/photos/notbrucelee/8016192302>

■ Note 17.1 Roadmap :

- Des exercices.
- Les extensions à la fin.

Sources

- *Algorithms* Robert Sedgewick, Kevin Wayne
- *Éléments d'algorithmique* D. Beauquier, J. Berstel, Ph. Chrétienne
- *125 Problems in Text Algorithms with Solutions* Maxime Crochemore, Thierry Lecroq, Wojciech Rytter

I Recherche dans un texte

I.1 Principe de la recherche

On s'intéresse ici au problème suivant :

Problème - RECHERCHETEXTE

- Entrées :
 - ★ une chaîne de caractère s sur l'alphabet Σ
 - ★ un autre chaîne de caractère m sur ce même alphabet appelé *motif* et de longueur plus petite que s
- Sortie : un résultat partiel correspondant à l'indice de la première occurrence du motif dans la chaîne s'il est présent.

La différence fondamentale entre ce problème et celui de la recherche d'un sous-tableau dans un tableau est le fait qu'on considère un alphabet fini et dont le nombre d'éléments est le plus souvent négligeable par rapport à la taille des chaînes de caractères. Cela permet d'effectuer des optimisations qui ne sont pas sans rappeler les tris linéaires comme le tri par comptage.

On parle alors d'algorithmique du texte pour désigner des algorithmes tirant partie de cette contrainte sur les données. La plupart des algorithmes que l'on présente peuvent ainsi s'adapter aisément au cas de tableaux dont les éléments sont pris dans un ensemble fini de petit cardinal.

Avant d'entamer ce chapitre, remarquons qu'il existe, outre l'alphabet usuel, trois alphabets très importants :

- celui des caractères ASCII usuels
- celui contenant les deux éléments 0 et 1, ce qui permet de travailler sur des recherche en binaire.
- et enfin, très important pour la biologie, l'alphabet à quatre lettres A, T, G et C correspondant aux bases d'un brin d'ADN et qui ouvre la porte à beaucoup d'applications en bio-informatique.

■ **Note 17.2** Il y aura sûrement des applications bio-info dans la partie programmation dynamique, faire le lien ici.

1.2 Algorithme naïf en force brute

Une solution naïve consiste à parcourir chaque position de s afin de tester si le motif est présent à partir de cette position.

indices	0	1	2	3	4	5	6	7	8
s	t	o	t	a	t	o	t	o	u

recherche à l'indice 0 t o t o
à l'indice 1 t o t o
à l'indice 2 t o t o
à l'indice 3 t o t o
à l'indice 4 t o t o
motif trouvé à l'indice 4

Cela donne l'implémentation assez directe suivante :

```
/* recherche_naive(m,s) recherche le motif m dans la chaîne
 * s et renvoie l'indice de la première occurrence s'il est présent
 * ou -1 sinon */
int recherche_naive(const char *m, const char *s)
{
    int n = strlen(s);
    int p = strlen(m);

    for (int i = 0; i <= n-p; i++)
    {
        int j;
        for (j = 0; j < p; j++)
        {
            if (s[i+j] != m[j])
                break;
        }
        if (j == p)
            return i;
    }

    return -1;
}
```

}

La complexité temporelle en pire cas de cet algorithme correspond au maximum de comparaisons. On peut naturellement en déduire par majoration une borne en $O(np)$ mais on peut remarquer qu'il est assez difficile d'obtenir un exemple concret, ce qui fait penser que ce pire cas est *rare*.

■ **Remarque 17.1** Considérons la chaîne $s = aa\dots a = a^n$ qui contient n fois la lettre a et le motif $m = a^{p-1}b$ qui contient $p - 1$ a et finit par un b . Dans l'algorithme, on va donc à chaque étape de la première boucle effectuer p itérations dans la seconde avant de se rendre compte que le motif n'est pas présent en comparant b et a . On a donc exactement $(n-p+1)p = \Theta(np)$ comparaisons et on retombe ainsi sur la complexité $O(np)$ pour ces exemples.

■

Ce qui va se passer dans une application usuelle de cet algorithme, c'est qu'au bout d'une ou deux comparaisons, on pourra invalider la position et passer à la suivante. On va alors avoir une complexité en $O(n+p)$ en considérant en plus la validation du motif dans le cas où il est présent. Ici $p \leq n$ donc $O(n+p) = O(n)$ mais c'est important de garder en tête cette complexité en $O(n+p)$ qu'on retrouvera car elle s'appliquera à des algorithmes où on effectue un prétraitement sur le motif pour l'appliquer ensuite sur plusieurs chaînes.

I.3 Algorithme de Boyer-Moore

Dans un premier temps, on va présenter la variante usuelle de cet algorithme appelée algorithme de Boyer-Moore-Horspool. On présentera ensuite l'algorithme de Boyer-Moore en tant que tel.

I.3.i Principe de Boyer-Moore-Horspool

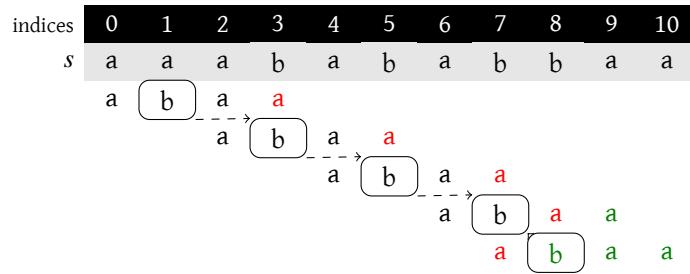
Le principe de l'algorithme de Boyer-Moore-Horspool est d'effectuer une recherche du motif comme précédemment mais en partant de la fin. On va alors tenter de trouver des suffixes de plus en plus grand du motif. Si on trouve ainsi le motif, on renvoie la position. Sinon, c'est qu'on a lu dans s un mot de la forme xm' où m' est un suffixe strict de m mais xm' n'en est pas un. Si x n'est pas présent dans m , alors on peut relancer la recherche juste après x dans s . Si x est présent dans m , on peut relancer la recherche en alignant ce caractère avec sa position la plus à droite dans m .

■ **Remarque 17.2** Il faut tenir compte différemment du dernier caractère du motif, car il n'est pas utile de le réaligner. On considère alors, quand elle existe, l'occurrence précédente de ce caractère.

■

On obtient ainsi une stratégie de saut qui en cas d'échec relance la recherche plus loin.

Voici un premier exemple où on effectue une recherche de $abaa$ dans le mot $aabababbaa$. Cette stratégie a permis d'éviter une recherche inutile à partir de l'indice 1.



1.3.ii Implémentation par table de saut

Pour réaliser ces sauts, on construit une table `droite` indexée par Σ et telle que $\text{droite}[c]$ indique l'indice de l'occurrence la plus à droite dans le motif m du caractère c , en ignorant le dernier caractère du motif.

Ainsi, dans l'exemple précédent du motif `abaa`, on obtient la table suivante :

c	'a'	'b'	'c'	...
<code>droite[c]</code>	2	1	\emptyset	...

On a indiqué ici \emptyset quand un caractère de Σ n'est pas présent dans le motif, car il peut être présent dans s .

Cette table contient donc de l'ordre de Σ éléments. On peut la réaliser par un tableau direct de taille $|\Sigma|$ étant donné un ordre d'énumération. On peut aussi la réaliser par un dictionnaire, ce qui est plus économique en espace si le motif contient peu de lettres différentes. On a choisi ici, pour des raisons pédagogiques, de considérer la numérotation ASCII naturelle associées au caractère de cette table.

```

int taille_alphabet = 256; // on pourrait passer par un define

/* Calcule le tableau droite associé au motif
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation */
int *calcule_droite(char *motif)
{
    int *droite = malloc(sizeof(int) * taille_alphabet);
    int p = strlen(motif);

    memset(droite, -1, sizeof(int) * taille_alphabet);

    for (int i = 0; i < p-2; i++)
    {
        int j = p-2-i;
        char c = motif[j];
        if (droite[c] < 0)
            droite[c] = j;
    }

    return droite;
}

```

Afin d'implémenter l'algorithme lui-même, il est nécessaire de faire des calculs élémentaires mais précis pour déterminer le saut à effectuer. Si à la position $i + j$ on a un échec après avoir lu le caractère c où $\text{droite}[c]$ contient la valeur k .

- Si $k = \emptyset$, c'est que le motif ne pourra jamais être trouvé tant que ce caractère c sera présent. On relance donc la recherche juste après à l'indice $i + j + 1$.

0	1	2	3	4	5	6	7	8
a	b	b	a	a	d	a	c	a
d	a	c						
		d	a	c				

- Si $k \geq j$, cela signifie que c est présent plus à droite dans le motif, donc aligner cette occurrence ne permettrait pas d'avancer la recherche. Rien ne nous permet de savoir si c est présent ou non ailleurs dans le motif, on relance alors prudemment la recherche en $i + 1$.

0	1	2	3	4	5	6	7	8
a	c	a	c	a	d	a	c	a
c	a	c						
	c	a	c					

- Sinon, on veut aligner ce c avec le caractère correspondant du motif, si on relance à l'indice i' , on souhaite ainsi avoir $i' + k = i + j$ donc $i' = i + j - k$.

```

/* cherche motif dans chaine en utilisant la table de saut précalculée
 * droite. Renvoie l'indice de la première occurrence ou -1 s'il n'est pas
 * présent */
int recherche_BMH(char *motif, int *droite, char *chaine)
{
    int n = strlen(chaine);
    int p = strlen(motif);

    for(int i = 0; i <= n-p; )
    {
        bool present = true;
        for (int j = p-1; j >= 0; j--)
        {
            if (chaine[i+j] != motif[j])
            {
                int k = droite[chaine[i+j]];
                present = false;
                if (k < 0)
                    i = i + j + 1;
                else if (k < j)
                    i = i + j - k;
                else
                    i = i + 1;
                break;
            }
        }
        if (present)
            return i;
    }
}

```

```

    return -1;
}

```

1.3.ii.a Correction

Tout d'abord, remarquons que la terminaison ne pose pas de questions dans la mesure où on le nouvel indice auquel on relance la recherche est toujours strictement plus grand que le précédent.

Au sujet de la correction, il suffit de s'assurer que les indices écartés correspondent nécessairement à des recherches infructueuses. Sans perte de généralité, on peut supposer que la recherche s'effectue depuis le premier indice de s . Comme seul les sauts d'au moins deux indices sont ceux pour lesquels il est nécessaire de faire une preuve, cela correspond au cas où $m = m_1cm_2dm_3x$ et $s = s_1cm_3s'$ avec c, d et x des caractères, $d \neq c$ et c non présent dans m_2dm_3 .

Ainsi, toute recherche démarrant à des indices inférieurs échouera systématiquement, au plus tard, en comparant le caractère c de cm_3 avec un caractère du motif dans m_2dm_3 donc différent de c .

1.3.ii.b Complexité

Tout d'abord, on remarque que la table de saut se construit en $O(\max(|m|, |\Sigma|))$ pour un motif m sur un alphabet Σ .

Sans chercher à rentrer dans les détails, on peut raisonnablement penser **si l'alphabet contient assez de caractères** que les motifs auront peu de répétitions et qu'ainsi, les sauts seront presque toujours maximaux, ce qui permet d'obtenir de l'ordre de $\frac{n}{p}$ comparaisons où n est la longueur de la chaîne et p la longueur du motif.

Cependant, en pire cas, cet algorithme n'est pas meilleur que le précédent. Pour s'en convaincre, on va considérer un exemple proche de celui introduit par l'algorithme naïf. Si on cherche ba^{p-1} dans a^n à l'indice i , il est nécessaire d'attendre de comparer au caractère b pour constater un échec et devoir relancer l'algorithme à l'indice $i + 1$. On va donc faire ici aussi $(n - p + 1)p = \Theta(np)$ comparaisons.

La complexité temporelle en pire cas de Boyer-Moore-Horspool est donc de $O(np)$, même si, en pratique, elle est sous-linéaire.

■ **Remarque 17.3** Si l'alphabet contient peu de caractères, ce qui est le cas en particulier du binaire, il y a de grandes chances qu'on soit dans ce cas pire cas. Ainsi, Boyer-Moore-Horspool n'est pas adapté pour ce type de texte. ■

1.3.iii Principe de Boyer-Moore

Considérons le cas suivant de l'algorithme précédent : on cherche abbcabc dans cbacbbcab.

0	1	2	3	4	5	6	7	8	9
c	b	a	c	b	b	c	a	b	c
a	b	b	c	a	b	c			
a	b	b	c	a	b		c		
a	b	b	c	a	b	c	a	b	c

On remarque qu'en raison du fonctionnement de cet algorithme, on est forcé de faire de tous petits sauts et on est ramené à l'algorithme naïf. Cependant, après la première étape, on sait qu'on a lu un suffixe du motif bc qui est précédé d'un caractère a en sorte que bbc ne soit pas un suffixe du motif.

Il y a un autre endroit dans le motif où on peut trouver $*bc$ avec $*$ un autre caractère que a . On pourrait donc relancer la recherche en alignant cette occurrence de bc avec celle qu'on vient de lire. Cela revient à sauter directement à la dernière étape dans cet exemple :

0	1	2	3	4	5	6	7	8	9
c	b	a	c	b	b	c	a	b	c
a	b	b	c	a	b	c			
			a	b	b	c	a	b	c

Pour pouvoir réaliser ce décalage, il est nécessaire de calculer une nouvelle table en parcourant le motif pour identifier de telles apparitions de suffixes.

On peut aller plus loin en considérant également le plus long préfixe du motif qui soit un suffixe du suffixe considéré. Par exemple, pour le motif $bcabc$ on remarque que bc étant un préfixe, on peut effectuer un saut comme dans l'exemple suivant :

0	1	2	3	4	5	6	7	8
c	a	a	b	c	c	b	b	c
b	c	a	b	c				
			b	c	a	b	c	

I.3.iii.a Table des bons suffixes

■ **Note 17.3** Tout cela sera redéfini proprement plus tard dans le chapitre sur les langages. Je laisse cette partie en attendant pour que la présentation soit complète.

Il est nécessaire d'introduire des définitions précises pour formaliser la stratégie qu'on vient de présenter. Dans le contexte des langages, on parle plus souvent de mot que de chaîne de caractères, qui sont un type de données permettant de les représenter. Un mot sur l'alphabet Σ est donc une suite finie $a_1 \dots a_n$ de lettres dans l'alphabet. On note μ l'unique mot vide, c'est-à-dire ne contenant aucune lettre. L'ensemble des mots sur Σ est noté Σ^* . Si u et v sont des mots, uv est le mot obtenu par concaténation.

■ **Remarque 17.4** Σ^* muni de cette loi de composition a une structure proche de l'ensemble des entiers naturels \mathbb{N} muni de l'addition :

- on a : $\forall u, v, w \in \Sigma^*, u(vw) = (uv)w = uvw$, on dit que la loi est associative;
- elle possède un élément neutre μ : $\forall u \in \Sigma^*, \mu u = u\mu = u$.

On dit alors que Σ^* est un **monoïde**. Cette structure très simple est cruciale en informatique.

Définition I.1 Soit $u, v \in \Sigma^*$, on dit que v est :

- un **suffixe** de u s'il existe $w \in \Sigma^*$ tel que $u = vw$
- un **préfixe** de u s'il existe $w \in \Sigma^*$ tel que $u = vw$

Lorsque $w \neq \mu$, on parle de suffixe ou de préfixe **propre**.

On dit que v est un **bord** de u lorsque v est suffixe et préfixe propre de u .

■ **Exemple 17.1** Soit $u = abacaba$. $abac$ est un préfixe de u , $caba$ un suffixe et aba un bord.

Définition I.2 Soit $x = x_1 \dots x_n$ et u, v deux suffixes **distincts** de x . On dit que u et v sont des suffixes **disjoints** quand on est dans l'un des cas suivants :

- $u = x$
- $v = x$
- $u \neq x, v \neq x$ et $x_{|x|-|u|} \neq x_{|x|-|v|}$.

Des suffixes disjoints sont donc des suffixes précédés par des lettres différentes dans x . On définit de même la notion de préfixes disjoints.

On considère un motif $x = x_0 \dots x_{n-1}$ et on va reprendre, en la précisant, la description précédente. Se faisant, on va construire une table **bonsuffixe** appelée la **table des bons suffixes** du motif x et telle que, pour $i \in \llbracket 0, n-1 \rrbracket$, **bonsuffixe**[i] donne le nombre de positions dont on doit décaler le motif vers la droite pour relancer la recherche après la lecture du suffixe $x_{i+1} \dots x_{n-1}$.

Supposons qu'on vient de lire avec succès un suffixe propre u . Ainsi $x = x_0 \dots x_i u$ et on vient de lire dans la chaîne où on effectue la recherche au avec $a \neq x_i$.

- Soit il existe un autre suffixe buv de x où $b \neq x_i$ et alors on appelle bon suffixe pour u un tel suffixe de longueur minimale et on pose alors **bonsuffixe**[i] = $|v|$
- Sinon, on cherche v de longueur minimale tel que x soit un suffixe de uv et on pose également **bonsuffixe**[i] = $|v|$.

On remarque que si x est suffixe de uv et qu'on a également buv' suffixe de x , alors $|uv| = |u| + |v| \geq |x| \geq |buv'| \geq |u| + |v'|$ donc $|v| \geq |v'|$ ce qui permet de considérer le plus petit v sur l'ensemble des cas.

I.3.iii.b Table des suffixes

Afin de calculer efficacement **bonsuffixe** on va commencer par calculer la table des suffixes du motif, il s'agit de la table **suffixe** où **suffixe**[i] contient la longueur du plus long suffixe de x de la forme $x_j \dots x_i$. Ainsi, si on note S_i les suffixes de cette forme, on a :

$$\text{suffixe}[i] = \begin{cases} 0 & \text{si } S_i = \emptyset \\ \max \{ |s| \mid s \in S_i \} & \text{sinon} \end{cases}$$

Nécessairement, **suffixe**[$n-1$] = n car x convient.

■ **Exemple 17.2** Pour $x = bcabc$ on a :

i	0	1	2	3	4
suffixe [i]	0	2	0	0	5

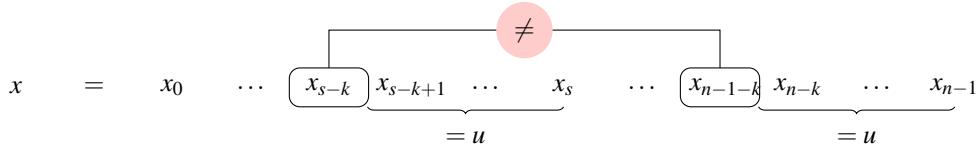
et pour $x = abbabba$:

i	0	1	2	3	4	5	6
suffixe [i]	1	0	0	4	0	0	7

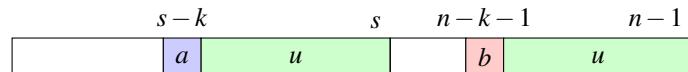
Il est possible de construire **suffixe** avec un simple parcours linéaire en tirant partie de l'information déjà calculée. Pour cela, on va remplir **suffixe** de droite à gauche.

A tout moment, on va conserver le meilleur suffixe rencontré, c'est-à-dire celui pour lequel on est allé le plus loin à gauche avant d'avoir un échec de comparaison. On note s la position la

plus à droite de ce suffixe et k sa longueur, il s'agit donc de $u = x_{s-k+1} \dots x_s$ et il y a eu un échec de comparaison en x_{s-k} . Par définition de **suffixe** on a **suffixe[s] = k**. Le mot x s'écrit alors :

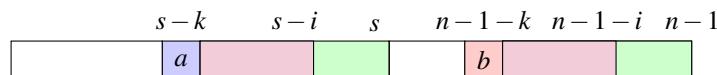


Ce qu'on peut représenter schématiquement ainsi :



Maintenant, on considère la position $s - i$ où $s > s - i > s - k$, cela signifie qu'on cherche un suffixe depuis une position interne au mot u de gauche. Le point clé permettant d'obtenir un algorithme linéaire est de remarquer que la situation est la même que dans le mot u de droite. Or, comme on procède de gauche à droite, on a déjà calculé la valeur correspondante **suffixe[n-1-i]**. Là, on a deux cas :

- soit quand on a cherché le plus grand suffixe à partir de $n - i$, on s'est heurté à une erreur de comparaison à la position $n - k$. Dans ce cas, on a **suffixe[n-1-i] = k - i** et on peut regarder, en partant de la position $s - k$, si on peut prolonger le suffixe finissant à la position $s - i$.



Pour effectuer ce prolongement, il suffit de comparer, caractère par caractère, vers la gauche en partant de la position $s - k$. On aboutira alors à une nouvelle position du suffixe finissant le plus à gauche qui finira en $s - i$.



Remarquons qu'il n'est pas nécessaire que $s - i - l \neq s - k$. C'est-à-dire que même si a ne permet pas de prolonger le suffixe déduit de la position $n - i$, on considère tout de même que la nouvelle position de référence est $s - i$. On en déduit également la valeur **suffixe[s-i] = l**.

- soit **suffixe[n-1-i] = p \neq k - i** et alors
 - ★ soit $p < k - i$, on a alors pour ce suffixe un échec dans u , ce qui limite de la même manière la valeur en $s - i$: **suffixe[s-i] = p**.
 - ★ soit $p > k - i$, donc on doit avoir un b après avoir le suffixe dans u depuis $s - i$ pour le prolonger, or, c'est impossible car il y a un $a \neq b$. Ainsi, le suffixe est limité par u : **suffixe[s-i] = k - i**.

Il reste à traiter le cas où $s - i \leq s - k$, ce qui revient à considérer qu'on a dépassé le précédent suffixe pouvant apporter une information. On procède donc naïvement pour trouver le plus grand suffixe depuis cette position.

On en déduit l'implémentation suivante :

```

/* Calcule le tableau suffixe associé au mot x
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation */
int *calcule_suffixe(char *x)
{

```

```

int n = strlen(x);
int *suffixe = malloc(sizeof(int) * n);
int plus_a_gauche = n-1;
int depart = -1;

memset(suffixe, -1, sizeof(int) * n);
suffixe[n-1] = n;

for (int j = n-2; j >= 0; j--)
{
    if (plus_a_gauche < j
        && suffixe[n-1-depart+j] != j-plus_a_gauche)
        suffixe[j] = MIN(suffixe[n-1-depart+j], j-plus_a_gauche);
    else {
        plus_a_gauche = MIN(plus_a_gauche, j);
        depart = j;
        while (plus_a_gauche >=0
               && x[plus_a_gauche] == x[n-1-j+plus_a_gauche])
            plus_a_gauche--;
        suffixe[j] = depart - plus_a_gauche;
    }
}

return suffixe;
}

```

■ **Remarque 17.5** Cette implémentation est optimisée par rapport à la description précédente en calculant directement sans introduire i ou k , et en fusionnant deux cas qui reviennent à dupliquer du code.

On donne ici le code maladroit qui correspond à la traduction exacte de la description précédente :

```

/* Calcule le tableau suffixe associé au mot x
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation */
int *calcule_suffixe_rep(char *x)
{
    int n = strlen(x);
    int *suffixe = malloc(sizeof(int) * n);
    int plus_a_gauche = n-1;
    int depart = -1;

    memset(suffixe, -1, sizeof(int) * n);
    suffixe[n-1] = n;

    for (int j = n-2; j >= 0; j--)
    {
        if (plus_a_gauche < j)
        {
            // on a j = depart - i avec
            int i = depart - j;
            // et plus_a_gauche = depart - k avec
            int k = depart - plus_a_gauche;

            if (suffixe[n-1-i] != k-i)
                suffixe[j] = MIN(suffixe[n-1-i], k-i);
            else {
                depart = j;
                while (plus_a_gauche >=0
                    && x[plus_a_gauche] == x[n-1-j+plus_a_gauche])
                    plus_a_gauche--;
                suffixe[j] = depart - plus_a_gauche;
            }
        } else {
            plus_a_gauche = depart = j;
            while (plus_a_gauche >=0
                && x[plus_a_gauche] == x[n-1-j+plus_a_gauche])
                plus_a_gauche--;
            suffixe[j] = depart - plus_a_gauche;
        }
    }

    return suffixe;
}

```

On remarque que dans ce code, `plus_a_gauche` ne peut que diminuer, on effectue donc au plus n itérations dans la boucle `while` pour tout l'algorithme. Donc, en considérant la boucle `for`, on effectue au plus $2n$ comparaisons de caractères : au plus une pour chaque itération de la boucle `for` pour voir si on entre dans le `while`, puis en tout au plus n avant de sortir du `while`.

L'algorithme qu'on a obtenue est bien linéaire en $|x|$. ■

I.3.iii.c Obtention de `bonsuffixe` à partir de `suffixe`

On reprend maintenant le calcul de `bonsuffixe[i]` dans le mot $x = x_0 \dots x_{n-1}$.

On cherche à obtenir des suffixe de la forme buv de x où $b \neq x_i$ et $u = x_{i+1} \dots x_{n-1}$ est un suffixe de x . Mais si $\text{suffixe}[k] = n - 1 - i$ cela signifie que ce suffixe est exactement u et qu'il est soit préfixe, soit précédé d'une lettre différente de x_i , sinon $n - 1 - i$ ne serait pas maximal.

On a donc

$$\begin{aligned}\text{bonsuffixe}[n-1-i] &= \min \{ n-1-k \mid \text{suffixe}[k] = n-1-i \} \\ &= n-1-\max \{ k \mid \text{suffixe}[k] = n-1-i \}\end{aligned}$$

On remarque qu'on peut ainsi faire croître k et poser :

$$\text{bonsuffixe}[n-1-\text{suffixe}[k]] = n-1-k$$

On a aura alors naturellement, à la fin de la boucle, la valeur minimale placée en dernier.

Reste à considérer les valeurs non remplies ainsi dans le tableau `bonsuffixe`. Elles correspondent aux positions i telles qu'il n'existe pas de suffixe de la forme buv . On doit donc chercher un mot uv de longueur minimale dont x est suffixe. Mais u étant un suffixe de x , cela revient à considérer les bords de x . La table `suffixe` permet également de détecter les bords : si $x_0\dots x_k$ est un bord c'est que $\text{suffixe}[k] = k+1$.

Soit $k < n-1$ maximal vérifiant cette condition. Pour tout $u = x_{i+1}\dots x_n$ suffixe de x , pour qu'il ait $x_0\dots x_k$ comme suffixe, il faut qu'il soit strictement plus long (sinon on est dans le cas précédent), donc que $n-i > k+1 \iff i < n-1-k$. Dans ce cas, x est alors suffixe de uv où $v = x_{k+1}\dots x_{n-1}$ donc $|v| = n-1-k$. Les k plus petits ne pourront alors que faire augmenter $|v|$, on peut ainsi poser $\text{bonsuffixe}[i] = n-1-k$.

On en déduit un remplissage en parcourant les k dans l'ordre décroissant de $n-2$ à 0 , tout en maintenant l'indice i de la prochaine valeur à remplir dans `bonsuffixe`. Dès qu'on détecte un bord, on place $n-1-k$ jusqu'à ce que $i \geq n-1-k$.

En sortie de boucle, il est possible que $i < n$ donc qu'il reste des valeurs à remplir. On remarque dans ce cas là que pour que x soit un suffixe de uv il faut que $v = x$. On a donc pour ces valeurs restantes $\text{bonsuffixe}[i] = n$.

Comme ce second cas est toujours plus long que le premier quand les deux se produisent en i , on implémente successivement les remplissages de sorte à obtenir la valeur minimum. On en déduit le programme suivant :

```
/* Calcule le tableau suffixe associé au mot x
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation */
int *calcule_bonsuffixe(char *x)
{
    int n = strlen(x);
    int *suffixe = calcule_suffixe(x);
    int *bonsuffixe = malloc(sizeof(int) * n);
    int suivant = 0;

    memset(bonsuffixe, n, sizeof(int) * n);

    for (int k = n-2; k >= 0; k--)
    {
        if (suffixe[k] == k+1) // bord
        {
            for (int i = suivant; i < n-1-k; i++)
                bonsuffixe[i] = n-1-k;
            suivant = n-1-k;
        }
    }

    for (int k = 0; k < n-1; k++)
        bonsuffixe[n-1-suffixe[k]] = n-1-k;
```

```

    free(suffixe);

    return bonsuffixe;
}

```

Il est facile de constater que cet algorithme est de complexité temporelle linéaire en $|x|$.

I.3.iii.d Algorithme de Boyer-Moore

On incorpore naturellement la table précédente à l'algorithme de Boyer-Moore en choisissant le meilleur décalage entre cette table et la stratégie précédente.

```

/* cherche motif dans chaine en utilisant les tables de saut précalculées
 * droite et bonsuffixe. Renvoie l'indice de la première occurrence ou -1 s'il n'est pas
 * présent */
int recherche_BM(char *motif, int *droite, int *bonsuffixe, char *chaine)
{
    int n = strlen(chaine);
    int p = strlen(motif);

    for(int i = 0; i <= n-p; )
    {
        bool present = true;
        for (int j = p-1; j >= 0; j--)
        {
            if (chaine[i+j] != motif[j])
            {
                int k = droite[chaine[i+j]];
                int dec = 1;
                present = false;
                if (k < 0)
                    dec = j + 1;
                else if (k < j)
                    dec = j - k;
                i = i + MAX(dec, bonsuffixe[j]);
                break;
            }
        }
        if (present)
            return i;
    }

    return -1;
}

```

Supposons que le motif est de longueur p , que la chaîne dans laquelle on recherche est de longueur n et que la taille de l'alphabet est une constante indépendante des entrées. La première partie de l'algorithme consiste à construire les tables de sauts, comme on l'a vu, elle est en complexité en temps et en espace en pire cas en $O(p)$.

On admet que l'algorithme Boyer-Moore complet, étant donné les deux tables de saut et d'autres modifications mineures non présentées ici, est en complexité temporelle en pire cas en $O(n)$.

Il est assez raisonnable de penser que soit $p \leq n$ quand on effectue une recherche, soit on compte chercher un même motif dans plusieurs textes et on réutilise ainsi les tables de sauts. Il n'est donc pas forcément très pertinent de parler de la complexité globale de l'algorithme,

mais lorsqu'on le fait, on dit qu'elle est en $O(p + n)$. On rappelle ici le rôle de l'addition dans les complexités qui fait référence à la succession de deux traitements, un en $O(p)$ suivi d'un en $O(n)$.

1.4 Algorithme de Rabin-Karp

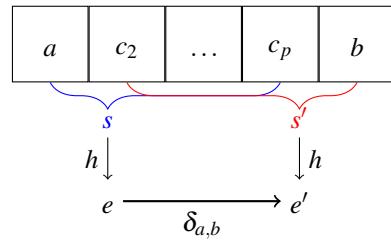
1.4.i Principe

L'algorithme de Rabin-Karp est un algorithme de recherche d'un motif dans un texte qui utilise une notion d'empreinte pour déterminer, en temps constant, si il est probable que la position actuelle corresponde à une occurrence du motif.

Pour cela, si on cherche un motif de longueur p sur l'alphabet Σ , on considère une **fonction de hachage** $h : \Sigma^p \rightarrow X$. Les éléments de l'ensemble X sont appelés des empreintes et on suppose que l'égalité entre deux empreintes se vérifie en temps constant contrairement à l'égalité dans Σ^p qui se vérifie en $O(p)$ dans le pire des cas. Le plus souvent, on choisit pour X un type entier machine.

■ **Note 17.4** Sûrement mettre ici des renvois vers la partie portant le plus sur la notion de fonction de hachage pour la définition la plus complète. ■

Bien qu'il soit normalement aussi coûteux de calculer l'image par h d'une sous-chaîne de longueur p que de tester l'égalité entre cette sous-chaîne et le motif, le point essentiel de l'algorithme de Rabin-Karp est d'utiliser une fonction de hachage permettant un calcul incrémental en temps constant :



Ici, on considère donc, pour $a, b \in \Sigma$, une fonction de mise à jour $\delta_{a,b} : X \rightarrow X$ telle que pour tout $c_2, \dots, c_p \in \Sigma$ on ait $\delta_{a,b}(h(ac_2 \dots c_p)) = h(c_2 \dots c_p b)$.

L'algorithme de Rabin-Karp procède alors ainsi pour chercher m de longueur p dans la chaîne $s = c_0 \dots c_{n-1}$ où $n \geq p$:

- calcul de $e_m = h(m)$ et $e = h(c_0 \dots c_{p-1})$.
- Pour i allant de 0 à $n - p$:
 - ★ Si $e_m = e$, on renvoie un succès pour la recherche à la position i si $m = c_i \dots c_{i+p-1}$
 - ★ si $i < n - p$ on met à jour l'empreinte $e \leftarrow \delta_{c_i, c_{i+p}}(e)$.

La complexité temporelle liée à la gestion des empreintes est donc en $O(n + p) = O(n)$ car $n \geq p$. Par contre, pour calculer la complexité liée à la recherche $m = c_i \dots c_{i+p-1}$, il est nécessaire d'estimer la proportion de faux positifs, c'est-à-dire de positions i telles que $e_m = e$ mais $m \neq c_i \dots c_{i+p-1}$. On va voir dans la partie suivante qu'on peut supposer qu'elle est négligeable, ce qui permet de considérer que l'algorithme de Rabin-Karp est linéaire.

1.4.ii Choix d'une fonction de hachage

Réaliser une bonne fonction de hachage est une question très complexe qui dépasse le cadre du cours d'informatique de MPI. Cependant, il est possible de réaliser ici une fonction de hachage répondant aux contraintes de Rabin-Karp assez facilement.

Pour cela, on considère que les caractères sont des entiers compris entre 0 et 255, ce qui correspond au type des caractères non signés sur un octet. On peut alors identifier une chaîne de longueur p avec un nombre entre 0 et $r^p - 1$ où $r = 2^8$, on note ainsi

$$P(c_0 \dots c_{p-1}) = \sum_{i=0}^{p-1} c_i r^{p-1-i} = c_0 r^{p-1} + c_1 r^{p-2} + \dots + c_{p-1}$$

On considère de plus un entier premier q et on pose $h(s) = P(s) \bmod q$ c'est-à-dire le reste de $P(s)$ dans la division euclidienne par q . On peut ainsi définir $\delta_{a,b}(e) = (r(e - ar^{p-1}) + b) \bmod q$.

Si on précalcule $r^{p-1} \bmod q$ il suffit d'un nombre d'opération constant, et indépendant de p , pour calculer la nouvelle empreinte à l'aide de $\delta_{a,b}$.

Le point essentiel est alors de déterminer un nombre premier q tel qu'il soit peu probable d'obtenir des faux positifs. Une analyse mathématique permet d'affirmer que chaque élément de $[|0; q - 1|]$ a de l'ordre de $\frac{r^p}{q}$ antécédents par h . Ainsi, si on choisit deux chaînes aléatoirement dans Σ^p , il y aura collision avec probabilité proche de $\frac{1}{q}$. En considérant q proche de la taille maximale pour le type entier considéré, on minimise donc cette probabilité.

■ Remarque 17.6 On peut également s'intéresser à des nombres q pour lesquels le modulo soit rapide à calculer. Un exemple classique est $q = 2^{31} - 1$ car on peut déduire la division euclidienne de a par q de l'écriture de a en base 2^{31} . En effet, si $a = \sum_{k=0}^n a_k 2^{31k}$ comme $2^{31} - 1 | 2^{31k} - 1$ pour $k \geq 1$, on a $2^{31k} \equiv 1 [q]$ et ainsi $a \equiv \sum_{k=0}^n a_k [q]$. On remarque que $a_k = (a >> 31k) \& 2^{31}$, on a alors soit $a_k < q$ et alors $a_k \bmod q = a_k$, soit $a_k = q$ et $a_k \bmod q = 0$. Il suffit donc de faire un masquage pour obtenir directement $a_k \bmod q = (a >> 31k) \& q$.

On obtient alors le programme suivant :

```
int64_t fastmod(int64_t a)
{
    int64_t s = 0;
    const int64_t q = 0x7fffffff;

    while (a > 0)
    {
        s = s + a & q;
        a = a >> 31;
    }

    if (s > q)
        return fastmod(s);
    if (s == q)
        return 0;
    return s;
}
```

Le programme suivant implémente naïvement les calculs de h et de $\delta_{a,b}$:

```
int64_t hash(int64_t r, int64_t q, char *s, int n)
{
    int64_t p = 1;
```

```

int64_t e = 0;

for (int i = n-1; i >= 0; i--)
{
    e = (p * s[i] + e) % q;
    p = (r * p) % q;
}

return e;
}

int64_t delta(int64_t r, int64_t q, int64_t rp,
              char a, char b, int64_t e)
{
    return (r * (e - rp * a) + b) % q;
}

```

1.4.iii Implémentation

Une implémentation directe de l'algorithme de Rabin-Karp est donnée dans le programme qui suit. On se sert ici du caractère paresseux du `&&` pour n'effectuer le test coûteux d'égalité des chaînes qu'en cas d'égalité des empreintes.

```

int rabin_karp(char *m, char *s)
{
    const int64_t r = 256;
    const int64_t q = 0x7fffffff;
    const int p = strlen(m);
    const int n = strlen(s);
    const int64_t rp = powmod(r,p-1,q);
    const int64_t me = hash(r,q,m,p);
    int64_t e = hash(r,q,s,p);
    for (int i=0; i <n-p+1; i++)
    {
        if (me == e && strncmp(m,(s+i),p) == 0)
            return i;
        if (i+p < n)
            e = delta(r,q,rp,s[i],s[i+p],e);
    }
    return -1;
}

```

Si on suppose qu'il est improbable d'obtenir un faux positif, il est possible de renvoyer un succès dès que les empreintes sont égales. L'avantage d'une telle version est alors d'être un algorithme sans retour sur les données. C'est-à-dire qu'il n'est pas nécessaire de garder en mémoire ou de réaccéder à un caractère.

1.4.iv L'algorithme originel de Rabin et Karp

Si on regarde l'article original de Rabin et Karp décrivant cette méthode, on peut être étonné du fait que la méthode précédemment décrite était considérée comme déjà connue dans la littérature par les auteurs. En fait, ce qu'ils décrivent et annoncent comme étant novateur est l'utilisation d'un algorithme probabiliste en choisissant aléatoirement une fonction de hachage à chaque lancement de l'algorithme. En pratique, il s'agit de choisir aléatoirement un nombre premier q parmi un ensemble précalculé de nombres premiers.

L'algorithme que l'on vient de décrire a un pire cas qui est très improbable car on considère que la probabilité d'un faux positif est à peu près de $1/q$, donc moins de $5 \cdot 10^{-10}$ pour $q = 2^{31} - 1$. Le problème ici est la notion de probabilité sur les entrées : est-on certain que l'algorithme recevra une entrée choisie uniformément ? Rabin et Karp parlent d'un *adversaire intelligent* qui aurait connaissance de la fonction de hachage choisie pour produire des entrées en pire cas. On pourrait ainsi imaginer une *attaque* sur serveur effectuant une recherche avec Rabin-Karp suite à l'entrée d'un utilisateur. Un adversaire pourrait construire une entrée en pire cas et tenter de surcharger le serveur en l'effectuant de manière répétée.

Pour bien mettre en lumière ce phénomène, nous allons ici construire, dans un cas très simple de fonction de hachage, une telle chaîne problématique. Pour cela, considérons la fonction de hachage précédemment décrite dans le cas de motif de taille 2, avec Σ contenant les lettres de a à z , $r = 26$ et $q = 17$. On considère une recherche du motif aa dont l'empreinte est 0, la même que celle des chaînes ar et ra . On peut donc considérer la chaîne $arar\dots ar$ qui produira un faux positif à chaque étape.

■ **Remarque 17.7** Détail des calculs. Ici on associe à a la valeur 0, ..., à z la valeur 25. On a donc

$$h(aa) = (0 \times 26 + 0) \mod 17 = 0$$

$$h(ar) = (0 \times 26 + 17) \mod 17 = 0$$

$$h(ra) = (17 \times 26 + 0) \mod 17 = 0$$

L'empreinte reste ainsi nulle tout au long de l'algorithme de Rabin-Karp et on a un faux positif à chaque itération. ■

II Compression

II.1 Principe

On s'intéresse ici à la compression parfaite d'un texte, c'est-à-dire, étant donné un alphabet fixé Σ , qu'on cherche à réaliser un couple de fonctions $comp, dec : \Sigma^* \rightarrow \Sigma^*$ telles que :

- pour tout mot $m \in \Sigma^*$, $dec(comp(m)) = m$
- pour la plupart des mots m qui correspondent aux données qu'on cherche à compresser, $|comp(m)| < |m|$.

■ **Remarque 17.8** Le fait que $dec \circ comp = id_{\Sigma^*}$ implique, comme on a pu le voir dans le cours de mathématique, que $comp$ est injective : deux mots différents ont nécessairement des images distinctes.

Si A et B sont deux ensembles finis tels que $|A| < |B|$, il n'existe pas de fonction injective de A dans B . Ainsi, si on note L_n les mots de Σ^* de longueur au plus n , il ne peut exister de fonction injective de L_n dans L_m où $n < m$.

Autrement dit : il est impossible d'espérer pouvoir compresser toutes les données de L_n . Si certains mots vont diminuer de longueur après compression, d'autres vont nécessairement augmenter.

Tout l'enjeu des algorithmes de compression parfaites est alors de diminuer les longueurs des mots qui nous intéressent. Par exemple, si on s'intéresse à des mots issus de textes en français, il est plus important d'arriver à compresser une phrase comme "ceci est un texte"

plutôt qu'une suite de caractères non signifiante comme "c2#\$%1ajdn //@#3d!fn". ■

II.2 Algorithme d'Huffman

La définition et la construction de l'arbre de Huffman ont été présentées au paragraphe Algorithme d'Huffman - Compression. On va s'intéresser ici au processus complet permettant de compresser et décompresser des fichiers avec cet algorithme.

II.2.i Calcul de la table d'occurrences

Par souci d'efficacité, on calcule une table d'occurrences pour l'ensemble des valeurs d'octets entre 0 et 255. Il suffit alors de parcourir le fichier pour incrémenter les valeurs correspondant aux octets lus.

ERROR: src/algorithmique/.../snippets/algorithmique/huffman.c does not exist

II.2.ii Sérialisation de l'arbre de Huffman

Afin de décompresser, il est nécessaire de connaître l'arbre de Huffman donnant le code préfixe. Pour cela, il faut stocker cet arbre dans le fichier comme une série d'octet, on parle de *sérialisation*. Cette notion sera prolongée dans le chapitre FIXME.

On choisit ici la représentation récursive `repr(a)` de l'arbre `a` définie ainsi :

- Si `a = Noeud(g,d)`, `repr(a) = 0 repr(g) repr(d)`
- Si `a = Feuille(c)`, `repr(a) = 1 c.`

■ **Exemple 17.3** Si `a = Noeud(Feuille 42, Noeud(Feuille 16, Feuille 64))`, on obtient la suite d'octets : 0 1 42 0 1 16 1 64. ■

La lecture et l'écriture de la sérialisation s'effectue alors simplement par récurrence :

ERROR: src/algorithmique/.../snippets/algorithmique/huffman.c does not exist

II.2.iii Écriture dans un fichier un bit à la fois

■ **Note 17.5** À déplacer éventuellement dans une partie spécifique sur la gestion de fichiers. ■

Le propre de l'algorithme de Huffman est d'associer à chaque caractère un codage binaire de longueur variable. Afin de pouvoir écrire ce codage dans un fichier, il est nécessaire de grouper les bits par paquet de huit (octet en français, byte en anglais).

Ainsi, par exemple, si on a le codage suivant :

c	'a'	'b'	'c'
<hr/>			
code(c)	0	100	101

et qu'on doit encoder "abbaca", on obtient le mot binaire 010010001010 qu'on complète avec des 0 à la fin et qu'on sépare en octets : 01001000 10100000. On obtient donc les deux octets, convertis en décimal, 72 et 160. Ce sont eux qu'on va écrire dans un fichier.

Une technique usuelle pour cela est de garder un accumulateur qui correspond à l'octet en train d'être construit ainsi que le nombre de bits qui ont été accumulé. Dès qu'on accumulé 8 bits, on peut construire l'octet, l'écrire dans le fichier, puis reinitialiser ces variables.

Quand on rajoute un bit b à l'accumulateur, on veut passer de $acc = b_1 \dots b_k$ à $b_1 \dots b_k b = 2acc + b$.

On en déduit l'implémentation assez directe suivante :

ERROR: src/algorithme/.../.../snippets/algorithme/bitpacking.c does not exist

Il reste à traiter la question des zéros finaux, si l'accumulateur contient k bits au moment de la fermeture du fichier, où $0 < k < 8$, il faut ajouter $8 - k$ zéros. On appelle cela du *padding* de l'anglais pour rembourrage. Ici, cela correspond à faire un décalage binaire vers la gauche d'autant (*shift left* en anglais). Comme il sera nécessaire de se souvenir que ces zéros ne sont pas significants à la lecture, on rajoute un octet final contenant cette valeur k .

On obtient alors la fonction de fermeture de fichier suivante :

ERROR: src/algorithme/.../.../snippets/algorithme/bitpacking.c does not exist

■ **Remarque 17.9** Une autre possibilité consiste à ajouter un entier décrivant la taille des données non compressées. C'est d'ailleurs parfois un problème avec les formats car si la taille est stockée sur 4 octets cela limite la taille d'un fichier pouvant être compressé.

Pour la lecture, on procède de même en faisant attention à deux points :

- on va lire les bits dans l'octet de la gauche vers la droite, c'est-à-dire du bit de poids le plus fort au bit de poids le plus faible. Ainsi, si l'accumulateur contient $acc = b_1 \dots b_8$, il suffit de faire un *et* bit à bit avec $b10000000=0x80=128$ pour obtenir $acc \& 0x80 = b_1 0 \dots 0$ donc un nombre qui vaut 0 si et seulement si $b_1 = 0$. Après avoir effectué cette lecture, il suffit de décaler vers la gauche en multipliant l'accumulateur par 2 : $2acc = b_2 \dots b_8$.
- on doit tenir compte des zéros finaux, pour ça, on a besoin de savoir qu'on est en train de lire le dernier caractère du fichier. On calcule donc la taille du fichier à son ouverture et on test si l'octet lu est l'avant-dernier, auquel cas on lit le dernier octet et on diminue d'autant le nombre de bits significants dans l'accumulateur.

On obtient alors le programme suivant pour la lecture :

ERROR: src/algorithme/.../.../snippets/algorithme/bitpacking.c does not exist

II.2.iv Compression d'un octet

Pour pouvoir compresser un octet, il est nécessaire d'obtenir le chemin qui mène jusqu'à la feuille dont il est l'étiquette dans l'arbre de Huffman. Pour cela, on commence par calculer l'ensemble des chemins de l'arbre de Huffman sous forme d'une table à 256 entrées qui contient le chemin associé à un octet s'il est présent dans l'arbre ou un chemin vide sinon. On parlera de représentation plate de l'arbre de Huffman.

Il suffit de faire un parcours exhaustif de l'arbre (FIXME référence aux parcours d'arbres) pour réaliser cette table :

ERROR: src/algorithme/.../.../snippets/algorithme/huffman.c does not exist

Afin de compresser un octet, on va donc aller lire le chemin dans cette table puis écrire le mot binaire correspond grâce aux fonctions d'écriture bit à bit :

ERROR: src/algorithme/.../.../snippets/algorithme/huffman.c does not exist

■ **Remarque 17.10** A chaque fois qu'on va compresser un octet, on va parcourir la liste correspondant à son chemin. Comme les chemins les plus longs sont les moins fréquents, cela ne

pose pas vraiment de problèmes.

Cependant, il est possible d'optimiser cela en ne stockant pas le chemin mais la fonction d'écriture elle-même. Ainsi, on ne va plus stocker des listes de booléens mais des fonctions du type `Bitpacking.out_channel_bit -> unit` qui vont réaliser l'écriture compressé de l'octet correspondant.

Au cours du parcours de l'arbre, on maintient une fonction correspondant à l'écriture du préfixe du chemin `chemin`. Si on l'appel est effectué sur un noeud, on remplace `chemin` par la fonction qui appelle `chemin` puis écrit le bit correspondant au côté gauche ou droit.

Cela correspond au programme suivant :

⌚ ERROR: `src/algorithmique/.../.../snippets/algorithmique/huffman.c` does not exist

Cette représentation des chemins partiels par des fonctions est très classique dans le style de programmation fonctionnelle par passage de continuations.



II.2.v Décompression d'un octet

Pour décompresser un octet, il suffit de parcourir l'arbre de Hufmann en lisant bit à bit le fichier compressé en descendant à gauche ou à droite selon que le bit lu soit 0 ou non. Dès qu'on arrive sur une feuille, on écrit dans le nouveau fichier le caractère correspondant.

⌚ ERROR: `src/algorithmique/.../.../snippets/algorithmique/huffman.c` does not exist

II.2.vi Compression et décompression de fichiers

En mettant bout à bout l'ensemble des fonctions, on obtient la fonction suivante qui réalise la compression complète d'un fichier :

⌚ ERROR: `src/algorithmique/.../.../snippets/algorithmique/huffman.c` does not exist

On obtient de même la fonction de décompression suivante :

⌚ ERROR: `src/algorithmique/.../.../snippets/algorithmique/huffman.c` does not exist

■ **Exemple 17.4** En compressant ainsi l'intégrale de Proust, on passe de 7543767 octets à 4249758 octets. A titre de comparaison, l'outil unix `zip` permet d'obtenir un fichier de 2724213 octets.



■ **Remarque 17.11** Le format de fichier présenté ici est rudimentaire. Les formats usuels sont en général plus complexes pour gérer

- l'identification : c'est-à-dire pouvoir déterminer qu'un fichier est un fichier compressé par un certain programme. Les fichiers `zip` commencent ainsi par les deux lettres `PK`.
- l'extensibilité : il est possible qu'on souhaite changer le format de sérialisation de l'arbre, ou même l'algorithme. En rajoutant un système de version sur les différentes parties, on peut permettre de faire évoluer un type de fichier en préservant la compatibilité avec les versions précédentes.



II.3 Algorithme de Lempel-Ziv-Welch

L'algorithme d'Huffman est efficace, mais il présente un désavantage majeur : il nécessite de lire le contenu d'un fichier dans son intégralité pour pouvoir déterminer un code préfixe optimal. Il est toutefois possible de modifier l'algorithme pour lever cette limitation. Dans ce

paragraphe, nous allons plutôt étudier une autre technique de compression qui, bien que moins efficace en pratique que Huffman, se programme assez facilement et permet de compresser des flux plutôt que des fichiers. C'est-à-dire qu'on peut compresser et décompresser des données au fur et à mesure qu'elles sont transmises.

Il s'agit de l'algorithme de Lempel-Ziv-Welch, appelé communément compression LZW, et qui est une modification faite en 1984 par Welch de l'algorithme de LZ78 de Lempel et Ziv.

II.3.i Principe de la compression

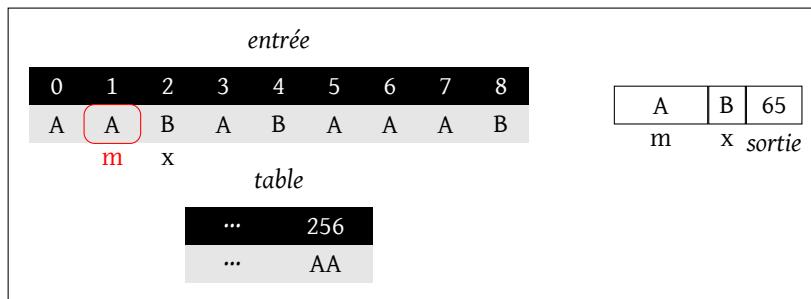
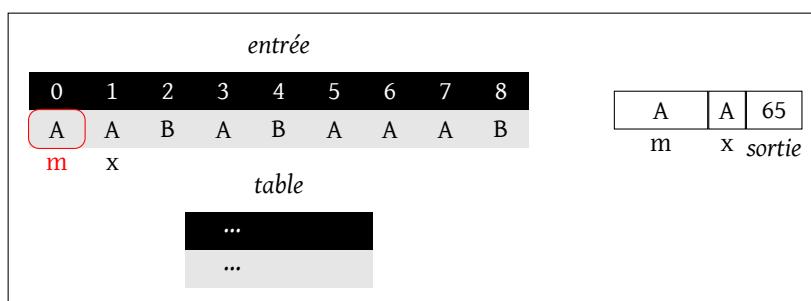
L'idée de l'algorithme LZW est de faire avancer une fenêtre sur le texte en maintenant une table des motifs déjà rencontrés. Quand on rencontre un motif déjà vu, on le code avec une référence vers la table et quand on rencontre un nouveau motif, on le code tel quel en rajoutant une entrée dans la table.

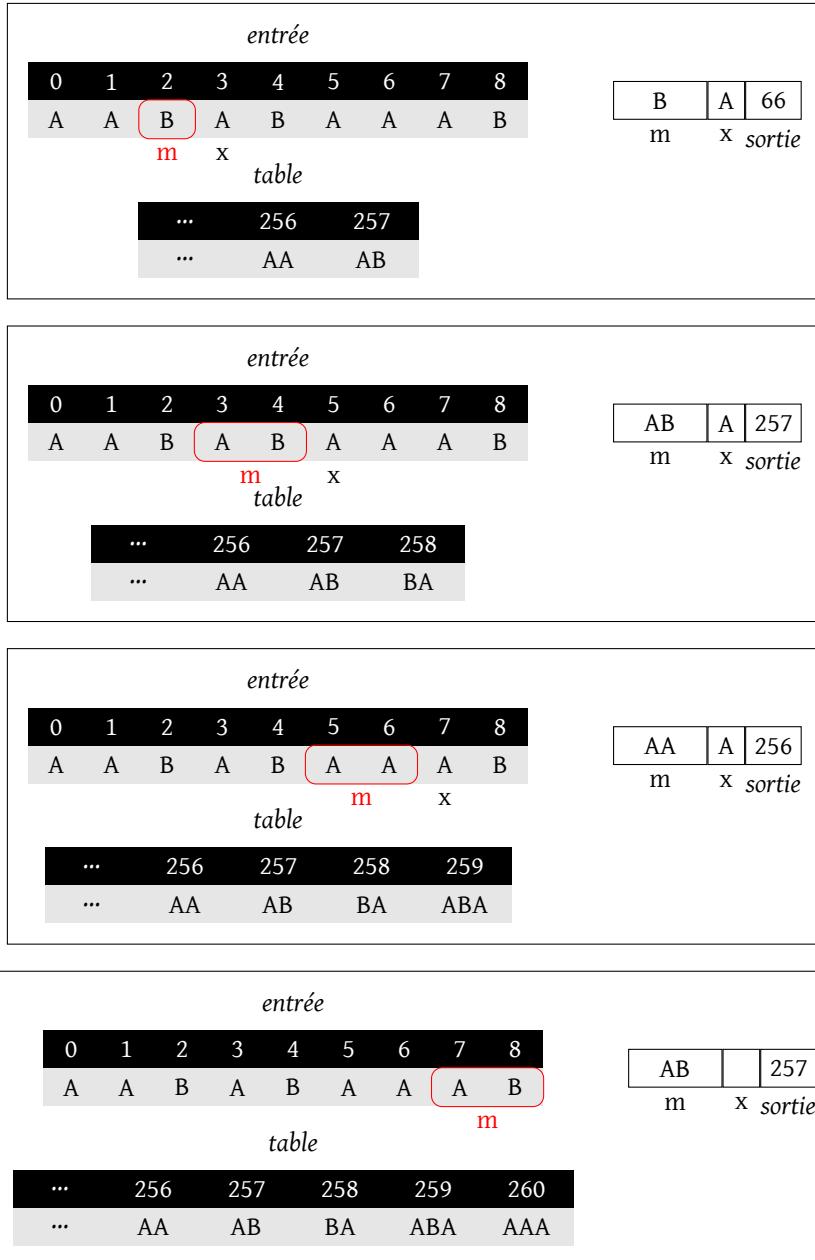
Pour la table, on peut utiliser un tableau dynamique de motifs (ref FIXME) dont la taille ne pourra pas dépasser 2^d éléments ou directement un tableau de 2^d valeurs optionnelles, dans la mesure où d est en général petit. Ainsi, on pourra référencer chaque motif avec un mot de d bits. Afin de pouvoir retrouver efficacement l'indice associé à un motif, on utilise une table de hachage (ref FIXME) réalisant l'inverse de la table.

L'algorithme procède alors ainsi pour compresser :

- on initialise la table avec une entrée pour chaque caractère, donc chaque octet, en considérant des caractères 8bit.
- on maintient une variable contenant le plus long suffixe m du texte lu qui soit présent dans la table, il est initialisé avec la première lettre du texte.
- on lit alors chaque caractère x :
 - ★ Soit mx est dans la table, et alors on remplace le motif courant par $m \leftarrow mx$
 - ★ Soit mx n'est pas dans la table, par construction m y est nécessairement on produit alors le code correspondant à m , on rajoute une entrée dans la table pour mx si elle contient moins de 2^d éléments et on repart de $m \leftarrow x$.
- quand tous les caractères ont été lus, on produit le code correspondant à m .

Voici les différentes étapes pour la compression de la chaîne AABABAAAB qui produit la suite d'entiers 65, 256, 66, 65, 258, 257 qui seront alors codés dans un fichier sur d bits. Les 256 premières entrées de la table ont été volontairement ignorées. On remarque juste que A correspond à l'index 65 et B à l'index 66.





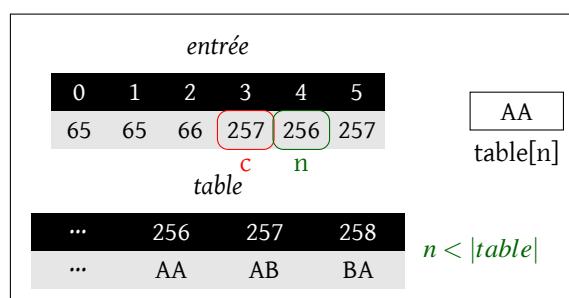
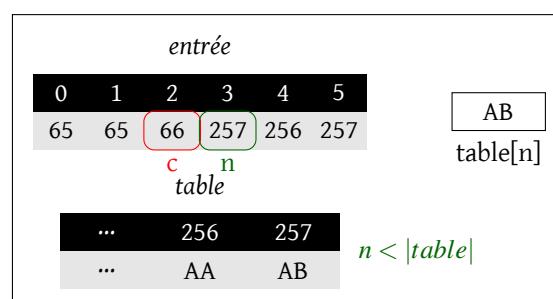
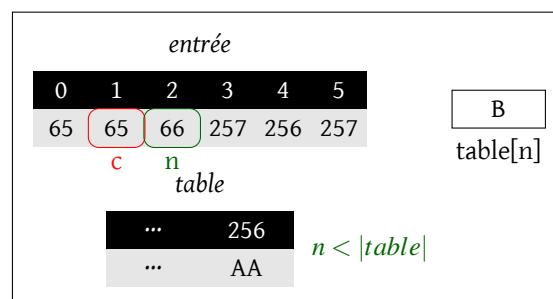
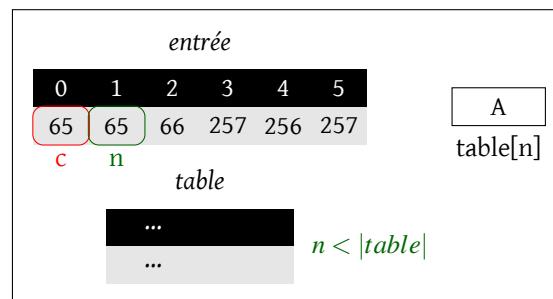
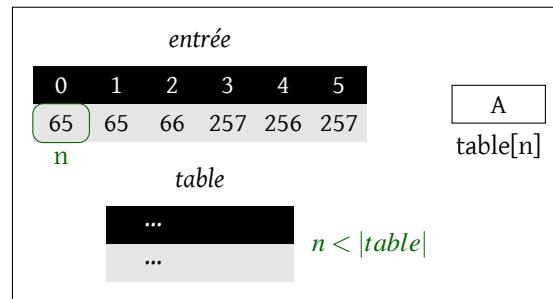
II.3.ii Principe de la décompression

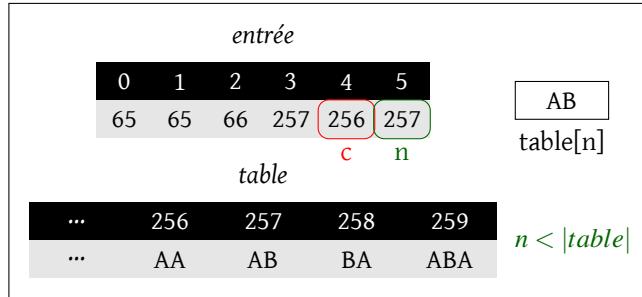
Pour décompresser, on effectue la procédure précédente en sens inverse. Cependant, il faut reconstruire la table en même temps qu'on lit le fichier compressé. Dans la majorité des cas, c'est assez immédiat. Pour le premier code lu, il s'agit forcément d'un référence à un des 256 caractères, donc on le reproduit. A partir du second code lu :

- on lit un code n où $n < |\text{table}|$ et $\text{table}[n] = xm'$, x est un caractère et m' un mot.
- on écrit xm' dans le fichier de sortie.
- on rajoute ensuite mx dans la table où c est le précédent code lu et $\text{table}[c] = m$

En faisant ainsi, on reproduit le processus de compression mais en remplaçant la table avec un temps de retard. En effet, si on reprend le principe exposé plus haut, une entrée pour mx est ajoutée dans la table quand on lit le caractère x et que le motif lu précédemment est m , on repart alors avec x pour motif lu. C'est exactement ce qu'on fait ici en tenant compte du premier caractère de $\text{table}[n]$. Contrairement à la compression, il est inutile ici de retrouver l'indice associé à un motif. On peut donc ignorer la table de hachage utilisée par la compression.

Voici les étapes de décompression de l'exemple précédent :



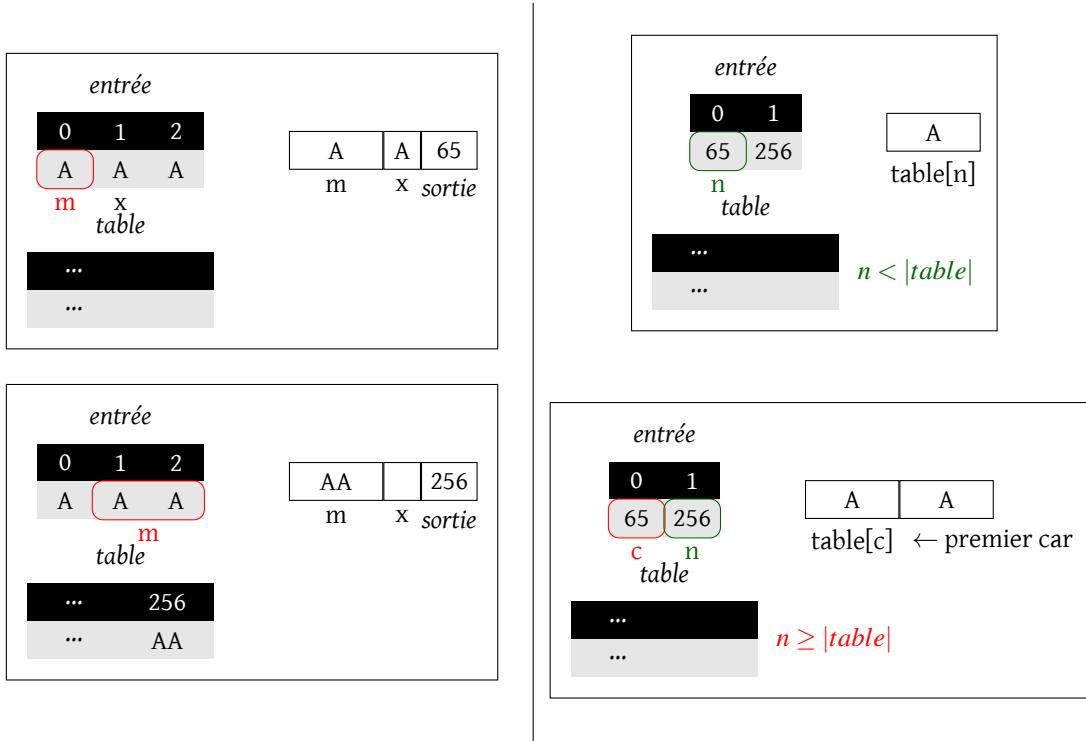


Il reste toutefois un cas à traiter, celui où $n = |\text{table}|$, c'est-à-dire quand on lit un code qui n'est pas encore présent dans la table. Pour comprendre ce cas, il est important d'identifier précisément quand il se produit dans le processus de compression. Comme on vient de le voir, on rajoute une entrée pour mx après avoir produit le code c correspondant à m . Pour que le code n ne soit pas présent dans la table, il faut donc que n corresponde à cette entrée mx . Or, quand on a compressé, on est reparti du motif x à ce moment là, donc nécessairement m commence par x . Cela signifie qu'on peut reconstruire $\text{table}[n]$ en décompressant avec mx où x est la première lettre de $\text{table}[c] = m$.

On en déduit alors la procédure complète suivante :

- on initialise la table avec une entrée pour chaque caractère, donc chaque octet, en considérant des caractères 8bit.
- on maintient une variable c contenant le dernier code lu qu'on initialise avec le premier code en produisant le caractère correspondant.
- pour chaque code n :
 - * Soit $n < |\text{table}|$ et $\text{table}[n] = xm'$ où x est un caractère, alors on écrit xm' en sortie
 - * Soit $n = |\text{table}|$ et alors on écrit en sortie $\text{table}[c]x$ où x est le premier caractère de $\text{table}[c]$
 - * Dans tous les cas, on remplace $c \leftarrow n$ et on ajoute $\text{table}[c]x$ à la table.

Il y a un exemple classique où on a besoin de traiter ce cas : celui où le même caractère est présent plusieurs fois en début de fichier. Comme LZW est utilisé pour compresser des formats d'images où les pixels sont des indices dans une palette de 256 couleurs avec une couleur pour la transparence, le cas où la même couleur est présente au début du fichier est fréquent. Voici ici un exemple de compression et de décompression pour AAA :



■ **Remarque 17.12** On peut pousser un peu plus l’analyse précédente afin d’identifier précisément les situations menant à ces cas problématiques. Comme on vient de le voir, il est nécessaire que le texte à compresser contienne un motif de la forme $xwxwx$ où w est un mot et x une lettre. Mais pour que le motif lu contienne xw au moment où on ajoute l’entrée xwx il faut que xw lui-même soit dans la table, ce qui signifie que chaque préfixe de xw est également dans la table au moment où on commence à le lire, mais que si le motif précédent est w' alors $w'x$ n’était pas dans la table.

On considère un mot $w = a_1 \dots a_n a$ où toutes les lettres sont distinctes, et on va essayer de construire un mot p tel que la compression de p permettra avoir $a_1, a_1 a_2, a_1 a_2 a_3, \dots, a_1 \dots a_n$ dans la table.

Si on lit $a_1 a_2$, on va ajouter a_1 puis $a_1 a_2$ dans la table. On continue alors en partant de a_2 comme motif, pour pouvoir ajouter $a_1 a_2 a_3$ à la table, il faut lire $a_1 a_2 a_3$ ensuite car à la lecture de a_1 , comme $a_2 a_1$ n’est pas dans la table on l’ajoute et m devient a_1 , puis $a_1 a_2$. Ainsi $a_1 a_2 a_1 a_2 a_3$ va ajouter $a_1, a_1 a_2, a_1 a_2 a_3$ dans la table. En continuant ainsi avec $a_1 a_2 a_3 a_4$, comme $a_3 a_1$ n’est pas dans la table, on va procéder de même.

Le mot $p = a_1 a_2 a_1 a_2 a_3 a_1 a_2 a_3 a_4 \dots a_1 a_2 \dots a_n$ va donc permettre d’ajouter tous les préfixes de m dans la table.

Maintenant, pour obtenir le cas précédent, il suffit de lire le texte $pwwa_1 x = pa_1 w_0 a_1 w_0 a_1 x$ où $w_0 = a_2 \dots a_n$ et x n’est pas l’un des a_i . En effet, on lit d’abord p ce qui permet d’ajouter tous les préfixes. Comme $a_n a_1$ n’est pas dans la table, on continue avec a_1 comme motif, on l’augmente jusqu’à lire $a_1 w_0$ puis on ajoute une entrée pour $a_1 m_0 a_1$ en ayant a_1 comme motif, et là on va lire $a_1 w_0 a_1 x$ et à la lecture de x , produire le code de $a_1 w_0 a_1$ qui est le dernier saisi dans la table.

Remarquons qu’on peut aussi ne considérer que $pwwa_1$ dans la mesure où la fin du fichier provoquera aussi l’écriture du code.

Si on considère $w = ABCD$, on a $p = ABABCABCD$ et on pourra ainsi compresser le texte

$$pwwA = pAw_0Aw_0A = \text{ABABCABCDABCDABCDA}$$

Après lecture de p on est dans la configuration suivante :

entrée																	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	B	A	B	C	A	B	C	D	A	B	C	D	A	B	C	D	A
<i>table</i>																	
...	256	257	258	259	260												
...	AB	BA	ABC	CA	ABCD												

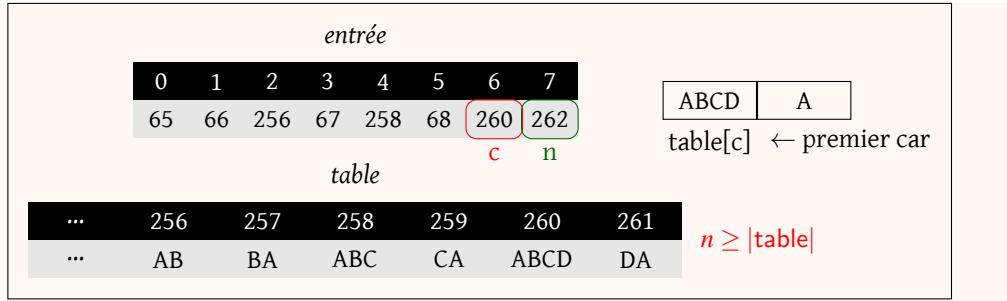
Puis après lecture de $wA = Aw_0A$:

entrée																	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	B	A	B	C	A	B	C	D	A	B	C	D	A	B	C	D	A
<i>table</i>																	
...	256	257	258	259	260	261											
...	AB	BA	ABC	CA	ABCD	DA											

Enfin, la lecture du reste du texte va déclencher l'ajout de ABCDA dans la table et l'écriture de son code.

entrée																	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	B	A	B	C	A	B	C	D	A	B	C	D	A	B	C	D	A
<i>table</i>																	
...	256	257	258	259	260	261	262										
...	AB	BA	ABC	CA	ABCD	DA	ABCDA										

A la décompression, on produira bien le texte correspondant avec le principe présenté plus haut :



■

II.3.iii Implémentation

Avant de commencer à implémenter l'algorithme, il est nécessaire de définir des fonctions de manipulation des entiers sur d bits et de lecture/écriture dans un fichier.

Tout d'abord, on définit des fonctions d'écriture d'entiers sous forme codée sur d bits à l'aide des fonctions vues précédemment :

ERROR: `src/algorithme/.../.../snippets/algorithme/bitpacking.c` does not exist

On implémente alors assez directement la compression :

ERROR: `src/algorithme/.../.../snippets/algorithme/lzw.c` does not exist

Et on procède de même pour la décompression :

ERROR: `src/algorithme/.../.../snippets/algorithme/lzw.c` does not exist

II.3.iv Impact de la longueur du code

Afin d'étudier l'impact de la longueur du code sur la taille des fichiers compressés, on considère deux fichiers :

- `proust.txt` contenant, en 7543768 octets, l'intégrale de à la recherche du temps perdu de Marcel Proust
- `code.py` contenant 8566 octets de code source Python

On note np la taille en nombre d'octets après compression du fichier `proust.txt` et tp le nombre d'entrées dans la table à la fin du processus. De même, on note nc et tc les valeurs respectives pour le fichier `code.py`.

On obtient alors les valeurs suivantes en fonction du nombre d de bits du code :

d	np	nc	tp	tc
8	7543768	8566	256	256
9	5056398	5709	512	512
10	4195124	4431	1024	1024
11	3864174	3948	2048	2048
12	3612505	4039	4096	2947
13	3434424	4376	8192	2947
14	3262145	4712	16384	2947
15	3131790	5049	32768	2947
16	2998639	5385	65536	2947
17	2682264	5722	131072	2947
18	2554323	6058	262144	2947

d	np	nc	tp	tc
19	2500314	6395	524288	2947
20	2557656	6731	1023317	2947

On constate qu'il est nécessaire d'avoir un texte riche pour bénéficier d'une grande longueur de code. Le fichier `code.py` ne contenant pas plus que 2947 motifs. Même si le fichier `proust.txt` en contient plus que les tailles considérées ici, il y a un compromis qui s'établit entre la richesse de la table et la taille du code. Ainsi, il semble que le fichier `proust.txt` soit compressé de manière optimale avec un code de longueur 19.

III Problèmes supplémentaires

- III.1 Transformation de Burrows-Wheeler
- III.2 Move to front
- III.3 La structure de données corde
- III.4 L'algorithme de Knuth-Morris-Pratt
- III.5 Extensions à l'analyse d'images

V

Systèmes

18	Gestion de la mémoire dans un programme compilé	281
I	Organisation de la mémoire	
II	Pointeurs	
III	Portée d'un identificateur	
IV	Piles d'exécution, variables locales et paramètres	
V	Allocation dynamique	



18. Gestion de la mémoire dans un programme compilé

■ Note 18.1 Roadmap :

- reprendre les exemples en C en utilisant des `int` suite au changement dans le programme.
- rajouter la présentation des pointeurs ici

■ **Remarque 18.1** Ce chapitre se concentre sur la manière dont un programme compilé gère la mémoire. Il est question, en particulier, de la notion de variable. Le modèle dans lequel on se place est celui du langage C où une variable est un emplacement mémoire.

I Organisation de la mémoire

■ **Note 18.2** Ici, je fais le choix d'une présentation assez informelle pour ne pas qu'elle soit trop liée à la réalité d'un compilateur en particulier.

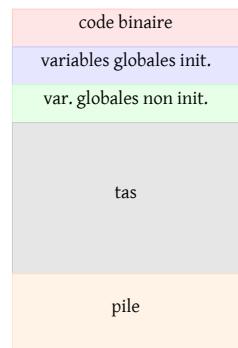
Un programme compilé gère la mémoire d'un ordinateur de deux manières très différentes

- statiquement : c'est le cas des variables locales ou globales définies dans le programme. Au moment de la compilation, le compilateur dispose de l'information suffisante pour prévoir de la place en mémoire pour stocker ces données.
- dynamiquement : c'est le cas des objets dont la taille n'est connue qu'à l'exécution et peut varier selon l'état du programme. C'est alors au moment de l'exécution que le programme va faire une demande d'allocation pour obtenir une place mémoire.

En terme d'allocation statique, on peut distinguer plusieurs types de mémoire :

- les variables globales initialisées qui seront stockées dans le binaire et placées en mémoire dans une zone spécifique chargée avec le binaire
- les variables globales non initialisées dont seule la déclaration sera dans le binaire et qui seront allouées, placées en mémoire et initialisées à 0 au moment du chargement du binaire
- les variables locales et les paramètres qui sont placés dans une pile afin de les allouer uniquement au moment de l'exécution du bloc ou de la fonction

L'allocation dynamique utilise une zone mémoire appelée tas dont une possible organisation est développée dans la partie sec. V.4.



Structure de la mémoire associée à un programme

Considérons le programme c suivant.

```

const int a = 42;
int b[] = { 1, 2, 3 };
int c;

int f(int x, int y)
{
    int z = x;
    z = z * y;
    return z;
}

int main(int argc, char **argv)
{
    const int d = 1664;

    c = f(a, d);

    return 0;
}
  
```

Il est possible d'observer la manière dont sa mémoire sera répartie en utilisant la commande objdump :

```

$ gcc -c memoire.c
$ objdump -x memoire.o

memoire.o:      file format elf64-x86-64
memoire.o
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

Sections:
Idx Name          Size   VMA           LMA           File off  Algn
  0 .text         00000053  0000000000000000  0000000000000000  00000040  2**0
  
```

```

CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1 .data      00000000 0000000000000000 0000000000000000 00000093 2**0
CONTENTS, ALLOC, LOAD, DATA
2 .bss       00000004 0000000000000000 0000000000000000 00000094 2**2
ALLOC
3 .rodata    00000014 0000000000000000 0000000000000000 00000098 2**3
CONTENTS, ALLOC, LOAD, READONLY, DATA
4 .comment   00000013 0000000000000000 0000000000000000 000000ac 2**0
CONTENTS, READONLY
5 .note.GNU-stack 00000000 0000000000000000 0000000000000000 000000bf 2**0
CONTENTS, READONLY
6 .eh_frame  00000058 0000000000000000 0000000000000000 000000c0 2**3
CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA

SYMBOL TABLE:
0000000000000000 l  df *ABS* 0000000000000000 orga.c
0000000000000000 l  d  .text 0000000000000000 .text
0000000000000000 l  d  .data 0000000000000000 .data
0000000000000000 l  d  .bss 0000000000000000 .bss
0000000000000000 l  d  .rodata 0000000000000000 .rodata
0000000000000000 l  d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l  d  .eh_frame 0000000000000000 .eh_frame
0000000000000000 l  d  .comment 0000000000000000 .comment
0000000000000000 g  O  .rodata 0000000000000004 a
0000000000000000 g  O  .data 000000000000000c b
0000000000000000 g  O  .bss 0000000000000004 c
0000000000000000 g  F  .text 000000000000001f f
0000000000000001f g  F  .text 0000000000000034 main

```

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
0000000000000042	R_X86_64_PLT32	f-0x0000000000000004
0000000000000048	R_X86_64_PC32	c-0x0000000000000004

RELOCATION RECORDS FOR [.eh_frame]:

OFFSET	TYPE	VALUE
0000000000000020	R_X86_64_PC32	.text
0000000000000040	R_X86_64_PC32	.text+0x000000000000001f

On retrouve dans les sections mémoire :

- .text contenant le programme binaire
- .data contenant les variables globales initialisées et non constantes
- .bss contenant les variables non initialisées
- .rodata contenant les variables globales initialisées mais constantes.

Ici, le schéma mémoire est un peu plus compliqué car une zone mémoire distincte est prévue pour les variables constantes initialisées pour des raisons de sécurité.

La pile et le tas sont automatiques et n'ont pas besoin de figurer dans le binaire, c'est pour cela qu'on ne les trouve pas dans la liste.

Dans cette description mémoire, on trouve la table des symboles qui décrit où vont se trouver

en mémoire certaines variables.

Nom de variable	Sorte de déclaration	Section mémoire
a	constante globale initialisée	rodata
b	globale initialisée	data
c	globale non init.	bss

La section suivante permettra de compléter le tableau en étudiant comment les paramètres et variables locales sont gérés. On remarque cependant qu'il n'y a pas de symboles pour ceux-ci. En effet, le nom des variables locales est *a priori* perdu après la compilation contrairement aux variables globales.

II Pointeurs

III Portée d'un identificateur

En ce qui concerne une variable dans un programme, on peut définir deux notions d'apparence assez similaire.

D'une part la portée d'un identificateur qui correspond au texte du programme :

Définition III.1 La portée d'un identificateur est définie par la zone du texte d'un programme dans laquelle il est possible d'y faire référence sans erreurs.

■ **Remarque 18.2** Dans le langage C, un identificateur peut être utilisé dès sa déclaration mais tant que la variable n'est pas initialisée, le comportement n'est pas spécifié et il faut considérer cela comme une erreur. Le compilateur produit ainsi un avertissement quand on utilise le paramètre `-Wall`.

Dans le cas d'une définition, il est ainsi possible de faire référence à l'identificateur dans l'expression de son initialisation : `int x = x`. Ce cas est pathologique et le fait qu'on compte la ligne de déclaration dans la portée ne devrait pas inciter à écrire ce genre de code qui produira, de toutes façons, une erreur avec les options `-Wall -Werror`.

Dans le programme :

```

1 int a = 1;
2
3 int f (int x)
4 {
5     int y = x + a;
6     return y;
7 }
8
9 int g()
10 {
11     int z = 3;
12     return z + f(z);
13 }
```

La portée des identificateurs est :

Identificateur	Portée
a	1-13
x	3-7
y	5-7
f	4-13
g	10-13
z	11-13

Pour une fonction, afin de pouvoir écrire des fonctions récursives, l'identificateur est utilisable dans le corps de la fonction.

Comme la portée est une notion associée aux identificateurs, elle est indépendante de la notion de variables. Si on considère le programme suivant :

```

1 int f()
2 {
3     int i = 3;
4
5     return i;
6 }
7
8 int g()
9 {
10    int i = 5;
11
12    return i+1;
13 }
```

L'identificateur *i* a pour portée les lignes 3-6 et 10-13.

Un autre phénomène plus complexe peut se produire quand on redéfinit un identificateur dans sa portée.

Considérons le programme suivant

```

1 int f()
2 {
3     int i = 3;
4
5     for (int j = 0; j < 3; j++)
6     {
7         int i = 4;
8
9         i += j;
10    }
11
12    return i;
13 }
```

Ici, l'identificateur *i* a pour portée les lignes 3-13 mais dans les lignes 7-10 il y a un phénomène dit de masquage où la première définition est cachée par la seconde.

L'identificateur associé à une variable globale a pour portée l'ensemble des lignes suivant sa déclaration.

■ **Remarque 18.3** En C, la portée d'un identificateur est **statique** : elle dépend uniquement du texte du programme au moment de la compilation.

En Python, la portée d'un identificateur est **dynamique** : elle peut dépendre de l'exécution d'un programme. Par exemple si on considère le programme Python

```
Python | if condition:
          x = 3
```

La portée de l'identificateur `x` dépend ici du fait que la `condition` soit réalisée ou non. ■

IV Piles d'exécution, variables locales et paramètres

On a vu qu'en raison de leur durée de vie, les variables globales étaient allouées dès le chargement du programme. Pour les variables locales ainsi que la mécanique des appels, on utilise une **pile**.

Cette pile d'exécution est représentée en mémoire par un tableau et un indicateur de fond de pile.

Le remplissage de ce tableau s'effectue souvent des adresses hautes vers les adresses faibles : on empile en faisant diminuer les adresses.

■ **Remarque 18.4** En fait, il existe des architectures où les adresses sont croissantes. Ce qui importe est que le tas et la pile aient des comportements opposés pour qu'ils puissent grandir dans la même zone mémoire. ■

Un compilateur peut faire le choix d'utiliser directement des registres processeurs pour les variables locales ou pour passer des paramètres à une fonction. Ici, pour simplifier, on va supposer que ce n'est pas le cas et que tout passe par la pile d'exécution.

■ **Remarque 18.5** Afin de pouvoir appeler une fonction dans une bibliothèque potentiellement compilée avec un autre compilateur, il est nécessaire d'avoir une convention d'appels de fonctions. Une telle convention est appelée une *interface applicative binaire* (Application Binary Interface).

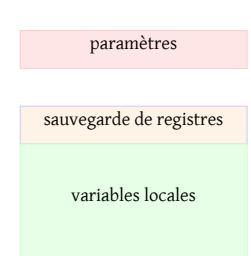
La convention System V AMD64 ABI qui est celle de Linux et macOS sur des architectures 64bits consiste à utiliser des registres entiers pour les six premiers arguments entiers ou pointeurs et des registres flottants pour les huit premiers arguments flottants. Les arguments suivants sont alors passés sur la pile (donc dès le septième entier/pointeur ou neuvième flottant).

La convention cdecl qui est assez répandue sur les architectures 32bits consiste à utiliser la pile. Par contre la valeur de retour est présente dans des registres comme pour la convention System V AMD64 ABI. ■

Lors d'un appel d'une fonction passant par la pile, on commence par empiler les paramètres (souvent de la droite vers la gauche) puis on empile l'adresse à laquelle doit revenir l'exécution une fois que la fonction aura terminé son exécution.

Au début de l'exécution de cette fonction, on place sur la pile l'adresse du fond de pile et on déplace celui-ci pour résERVER de la place pour les variables locales.

L'empreinte sur la pile d'un appel de fonction est appelée une structure pile (**stack frame** en anglais). La pile est alors organisée, depuis l'appel au point d'entrée du programme, par empilement et dépilement de structures piles.



Structure pile associée à un appel de fonction

Voici un exemple possible de l'état de la pile d'exécution lors de l'exécution d'un programme compilé avec la norme `cdecl` (Il suffit d'ajouter l'argument `-m32` pour compiler en 32bits).

```

1 int f(int a, int b)
2 {
3     int c = 3;
4     /* pile ici après l'appel en l13 */
5     c = c + b;
6     c = c * a;
7     return c;
8 }
9
10 int main()
11 {
12     int x = f(10,12);
13 }
14

```



A chaque appel de fonction, on va donc empiler une structure de pile complète, puis la dépiler à la sortie. Ce mécanisme est essentiel pour permettre la récurrence car il permet d'effectuer plusieurs appels d'une même fonction sans risquer que la mémoire utilisée lors d'un des appels interfère avec un autre. On comprend également les limites de la récursivité ici car cet empilement successif de structures de piles peut dépasser la taille maximale de la pile d'exécution : on parle alors de *dépassement de pile* ou **stack overflow** en anglais.

■ **Remarque 18.6** Il est possible de définir des variables locales qui soient situées au même emplacement mémoire pour tous les appels d'une fonction, c'est ce qu'on appelle des variables *statiques* dans le paragraphe précédent.

Ce mécanisme est essentiellement géré comme les variables globales et il ne sera pas développé dans la suite.

■ **Note 18.3** Je me demande s'il faudrait parler plus précisément de la manière dont la pile est gérée avec les registres `ebp/esp`. Mais ça ne me semble pas apporter grand chose ici.

V Allocation dynamique

Comme cela a été vu dans la section sec. I, il est possible d'allouer dynamiquement de la mémoire. Pour gérer cette allocation dynamique, on passe par une zone mémoire appelé le *tas* ainsi que par un mécanisme d'allocation et de libération de mémoire au niveau du système.

Pour l'utilisateur, cette gestion interne est transparente et on peut se contenter de considérer qu'il y a deux mécanismes :

- **l'allocation** mémoire où on demande à ce qu'une zone mémoire d'une certaine taille soit allouée
- **la libération** mémoire où on signale que la zone mémoire peut être récupérée par le système.

Naturellement, la mémoire d'un ordinateur étant finie, il est très important de libérer au plus tôt la mémoire non utilisée pour éviter d'épuiser la mémoire. Quand un programme ne libère pas toute la mémoire qu'il alloue, on parle de **fuite mémoire**. L'empreinte mémoire d'un tel programme peut alors croître jusqu'à rendre le programme ou le système inutilisable.

V.1 Allocation

Pour allouer une zone mémoire, on utilise la fonction `malloc` dans `stdlib.h` de signature :

```
void *malloc(size_t size)
```

Ici `size` indique le nombre d'**octets** à allouer et la fonction renvoie un pointeur vers le premier octet alloué. Comme la fonction ne connaît pas le type d'objets alloués, on utilise ainsi le type `void *`.

Ce type joue un rôle spécial et on peut changer directement le type de la valeur de retour sans rien avoir à écrire d'autre l'appel à `malloc` :

```
char *t = malloc(n);
```

■ Remarque 18.7 Dans le langage C++ qui peut être vu comme un successeur de C, il est obligatoire de préciser ici le nouveau type à l'aide d'un `transtypepage`. Pour convertir la valeur `x` vers le type `char *` on écrit alors `(char *)x`. Ainsi, pour allouer un tableau de `n` caractères, on utilisera :

```
char *t = (char *) malloc(n);
```

Bien que ce ne soit pas nécessaire en C, il est fréquent de rencontrer des programmes présentant de tels transtypages qui sont superflus mais corrects syntaxiquement en C.

Pour obtenir la taille à allouer, il peut être utile d'utiliser l'opérateur `sizeof` qui prend en entrée un type ou une valeur et renvoie sa taille. Ainsi si on peut allouer un tableau de `n` entiers non signés ainsi :

```
unsigned int *t = malloc( sizeof(unsigned int) * n );
```

et cet appel ne dépend de la taille prise par un `unsigned int` sur l'architecture.

Une autre raison de l'utilisation de `sizeof` est l'extensibilité. Par exemple, si on a un `struct point` représentant des points en 2D :

```
C | struct point {
C |     float x;
C |     float y;
C | };
```

on peut allouer un tableau de n points ainsi :

```
C | struct point *t = malloc( sizeof(struct point) * n );
```

Si jamais on change la structure pour représenter des points en 3D ainsi :

```
C | struct point {
C |     float x;
C |     float y;
C |     float z;
C | };
```

il sera inutile de changer le code d'allocation du tableau car `sizeof(point)` tiendra compte automatiquement du changement.

Si jamais une erreur empêche d'allouer la mémoire - ce qui peut être le cas s'il n'y a plus de mémoire disponible - le pointeur renvoyé par `malloc` a la valeur spéciale `NULL`.

■ Remarque 18.8 La zone mémoire renvoyée par `malloc` n'est pas initialisée. On ne peut pas supposer qu'elle soit remplie de la valeur 0. Il faut donc manuellement initialiser la mémoire après le retour de `malloc`.

V.2 Libération

Pour libérer la mémoire, on utilise la fonction `free` également présente dans `stdlib.h` et dont la signature est :

```
C | void free(void *ptr);
```

■ Remarque 18.9 Il est très important d'utiliser uniquement un pointeur obtenu précédemment par un appel à `malloc` et de ne pas l'utiliser plus d'une fois.

Le programme suivant provoque une erreur `free(): invalid pointer` à l'exécution mais est détecté par un avertissement du compilateur : `warning: attempt to free a non-heap object 'a'`.

```
#include <stdlib.h>

int main()
{
    int a;
    free(&a);
    return 0;
}
```

Le programme suivant alloue un tableau de deux `char` et appelle `free` sur l'adresse de la seconde case. En faisant cela, il n'y a pas d'avertissement car on appelle `free` sur un objet

qui est effectivement sur le tas. On obtient alors à nouveau une erreur à l'exécution `free(): invalid pointer`.

```
#include <stdlib.h>

int main()
{
    char *a = malloc(2);
    free(&(a[1]));
    return 0;
}
```

Le programme suivant libère deux fois la mémoire et provoque l'erreur `free(): double free detected in tcache 2` à l'exécution.

```
#include <stdlib.h>

int main()
{
    char *a = malloc(2);
    free(a);
    free(a);
    return 0;
}
```

V.3 Protection mémoire

Comme on l'a vu dans la première partie, quand un programme s'exécute il a un environnement mémoire constitué de plusieurs zones, parfois appelées **segments**, avec le droit d'écriture dans certaines d'entre elles.

Le système d'exploitation protège ainsi la mémoire et, au niveau matériel, l'unité de gestion de la mémoire connaît les adresses accessibles à un programme. En cas d'accès anormal, le matériel provoque une erreur qui remonte au système d'exploitation qui termine l'exécution du programme avec une erreur souvent intitulée `Segmentation fault`.

Voici quelques exemples commentés produisant des erreurs de type `segmentation fault` à l'exécution.

Lecture à l'adresse 0, ce qui provoque toujours une erreur.

Même problème avec une adresse inaccessible ou invalide.

```
int main()
{
    int *a = 0;
    return a[0];
}
```

Écriture dans une zone en lecture seule comme le segment du code.

```
int main()
{
    int *a = (int*) (&main);
    // a pointe sur le corps de la fonction main

    *a = 0;

    return 0;
}
```

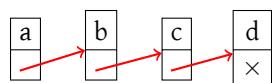
V.4 Réalisation d'un système d'allocation de mémoire

■ Note 18.4 Prérequis : listes chaînées

Afin de comprendre comment fonctionne le tas, et en particulier malloc et free, on va simuler ici ce comportement en allouant une grande plage de mémoire avec malloc et en gérant le découpage et l'allocation de celle-ci.

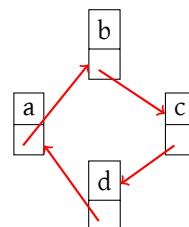
Pour gérer les blocs mémoires libres, on utilise une liste circulaire. Une liste circulaire est une liste chaînée avec un lien supplémentaire entre le premier et le dernier maillon, ce qui fait qu'on peut considérer n'importe quel maillon comme étant la *tête* de la liste.

Une liste chaînée :



Ici le dernier maillon comprend un pointeur qui ne pointe sur rien indiqué par \times , en pratique il a la valeur NULL

Une liste circulaire :



Le seul changement est donc de faire pointer le dernier maillon sur le premier. Le fait d'avoir préserver les valeurs dans les maillons permet ici de voir ce qu'est devenu le premier maillon, mais on peut accéder à cette liste par n'importe lequel de ces maillons.

Les noeuds de la liste circulaire de blocs libres auront pour valeur un triplet (`adresse`, `taille`, `libre`) qui indique qu'à l'adresse `adresse` il y a un bloc de `taille` octets et le booléen `libre` indique si ce bloc a été alloué ou non.

Pour cela on commence par définir une structure `bloc` et une fonction de création d'un bloc :

```
struct bloc {
    void *adresse;
    uint32_t taille;
    bool libre;
    struct bloc *suivant;
};
```

```

struct bloc *cree_bloc(void *adresse, uint32_t taille, bool libre)
{
    struct bloc *b = malloc(sizeof(struct bloc));
    b->adresse = adresse;
    b->taille = taille;
    b->libre = libre;
    return b;
}

```

On définit ensuite deux variables globales :

- `bloc_libres` qui va pointer sur un maillon de la liste circulaire des blocs
- `plage_memoire` qui pointe sur l'adresse de la plage mémoire que l'on va gérer et servira à la libérer en sortie de programme.

```

struct bloc *blocs_libres;
void *plage_memoire;

```

La fonction `creation_blocs_libres` permet de créer la liste circulaire avec un premier bloc qui pointe sur lui-même et qui correspond à l'adresse que l'on va placer dans `plage_memoire`.

```

void creation_blocs_libres(uint32_t taille_bloc_initial)
{
    plage_memoire = malloc(taille_bloc_initial);
    blocs_libres = cree_bloc(plage_memoire, taille_bloc_initial, true);
    blocs_libres->suivant = blocs_libres; // boucle initiale
}

```

Pour libérer la liste à la sortie du programme, on définit la fonction `destruction_blocs_libres` qui présente ainsi le parcours usuel d'une liste circulaire : on procède comme pour une liste chaînée classique mais, au lieu d'utiliser un test `bloc_courant->suivant == NULL` pour l'arrêt, il faut se souvenir du premier bloc et tester pour voir si on est revenu au point de départ. On n'oublie pas de libérer l'espace `plage_memoire` à la fin.

```

void destruction_blocs_libres()
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    // on boucle pour libérer chaque maillon
    while (true) {
        struct bloc *bloc_suivant = bloc_courant->suivant;
        free(bloc_courant);
        if (bloc_suivant == premier_bloc) return;
        bloc_courant = bloc_suivant;
    }

    // on libère la plage mémoire initiale
    free(plage_memoire);
}

```

Pour allouer t octets, on parcourt la liste des blocs jusqu'à trouver un bloc b libre de taille $b.t$ telle que $b.t \geq t$. Si un tel bloc n'existe pas, on renvoie le pointeur `NULL` signe d'un échec d'allocation. Sinon, on indique que le bloc est occupé, on va renvoyer l'adresse du bloc obtenu

mais, si $b.t > t$ on insère après b un nouveau bloc libre de taille $b.t - t$. Dans tous les cas, on fait pointer la liste des blocs libres vers le bloc qui suit b , qui est peut-être le bloc nouvellement créé et a de grandes chances d'être libre.

Ce mécanisme est implementé dans la fonction allocation :

```
void *allocation(uint32_t taille)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    while (!bloc_courant->libre || bloc_courant->taille < taille)
    {
        bloc_courant = bloc_courant->suivant;
        if (bloc_courant == premier_bloc)
        {
            // Retour au point de départ : échec d'allocation
            return NULL;
        }
    }

    // bloc_courant pointe sur un bloc libre de bonne taille

    void *adresse = bloc_courant->adresse;
    bloc_courant->libre = false;
    if (bloc_courant->taille > taille) {
        // on le sépare en deux pour récupérer la place
        struct bloc *bloc_libre = cree_bloc(adresse+taille,
                                             bloc_courant->taille-taille, true);
        bloc_courant->taille = taille;
        bloc_libre->suivant = bloc_courant->suivant;
        bloc_courant->suivant = bloc_libre;
    }

    // On pointe sur le bloc suivant qui est sûrement libre
    blocs_libres = bloc_courant->suivant;

    return adresse;
}
```

Pour libérer un bloc, on parcourt la liste jusqu'à trouver le bloc correspondant à l'adresse à libérer et on indique que le bloc est libre. Ici, il y a deux assert permettant de s'assurer que l'adresse est bien celle d'un bloc et que le bloc n'a pas déjà été libéré.

```
void liberation(void *adresse)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    while (bloc_courant->adresse != adresse)
    {
        bloc_courant = bloc_courant->suivant;
        // adresse invalide
        assert(bloc_courant != premier_bloc);
    }
```

```
// pas de double libération
assert(!bloc_courant->libre);

// on libère l'adresse
bloc_courant->libre = true;
}
```

Voici un premier programme de test de ces fonctions qui alloue 10 octets puis effectue plusieurs allocations. L'allocation de c échoue car il n'y a plus de place libre.

```
1 int main(void)
2 {
3     creation_blocs_libres(10);
4
5     uint8_t *a = allocation(5);
6     uint8_t *b = allocation(3);
7     uint8_t *c = allocation(3);
8     liberation(b);
9     uint8_t *d = allocation(2);
10
11    printf("Allocation a:%p b:%p c:%p d:%p\n",
12          (void *)a, (void *)b, (void *)c, (void *)d);
13
14    destruction_blocs_libres();
15
16    return 0;
17 }
```

Ce programme affiche alors

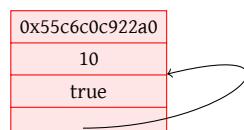
Allocation a:0x55c6c0c922a0 b:0x55c6c0c922a5 c:(nil) d:0x55c6c0c922a5

Voici l'évolution de la liste circulaire en présentant les maillons sous la forme :

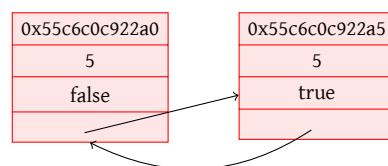
adresse
taille
libre
suivant

L'évolution de la liste des blocs entre les lignes 3 et 9 est alors :

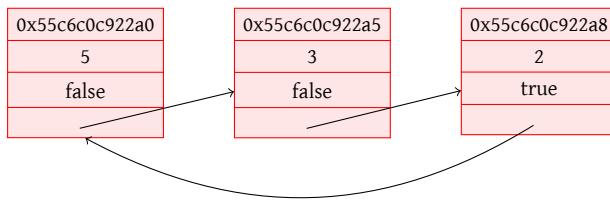
- Ligne 3



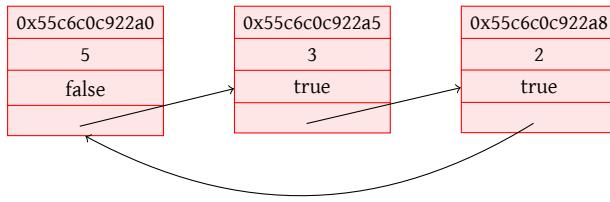
- Ligne 5



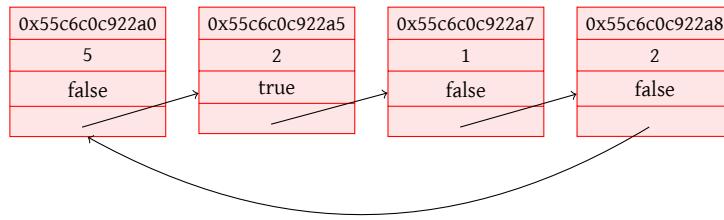
- Ligne 6



- Ligne 8



- Ligne 9



Cette méthode d'allocation a un défaut majeur : elle fragmente l'espace libre. Dans le programme suivant, il sera impossible d'allouer b car la liste circulaire contient deux blocs libres de 5 octets et non une bloc libre de 10 octets.

```
creation_blocs_libres(10);
c uint8_t *a = allocation(5)
liberation(a);
uint8_t *b = allocation(10);
```

■ **Remarque 18.10** On peut observer ce phénomène sur le diagramme précédent à la ligne 8 où deux blocs contigus sont libres et pourraient être fusionnés en un unique bloc de 5 octets. ■

On peut éviter cela en effectuant une phase de coalescence des blocs libres à la libération. En vertu de la nature de la liste circulaire, il est nécessaire de fusionner un bloc libre avec les blocs suivants. En utilisant une liste circulaire doublement chaînée, on pourrait également fusionner avec les blocs précédents.

Pour cela on change la fonction liberation ainsi :

```
void liberation(void *adresse)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;
```

```
while (bloc_courant->adresse != adresse)
{
    bloc_courant = bloc_courant->suivant;
    // adresse invalide
    assert(bloc_courant != premier_bloc);
}

// pas de double libération
assert(!bloc_courant->libre);

// on libère l'adresse
bloc_courant->libre = true;

premier_bloc = bloc_courant;
bloc_courant = bloc_courant->suivant;

while (bloc_courant != premier_bloc && bloc_courant->libre)
{
    struct bloc* actuel = bloc_courant;
    premier_bloc->taille += bloc_courant->taille;
    bloc_courant = bloc_courant->suivant;
    free(actuel); // on libere le bloc inutile
}

premier_bloc->suivant = bloc_courant;
}
```