

# Algorithmes – Notes de cours

F. Popineau

Ce document rassemble des notes de cours.  
Sa rédaction n'est pas achevée.  
Merci de rapporter toute erreur.

# Table des matières

<b>Table des matières</b>	<b>ii</b>
<b>I Introduction</b>	<b>1</b>
<b>1 Ressources</b>	<b>3</b>
<b>2 Complexité des algorithmes</b>	<b>5</b>
2.1 Problème et définition . . . . .	5
2.2 Pseudo-code . . . . .	6
2.3 Complexité asymptotique . . . . .	7
2.4 Règles de calcul . . . . .	10
2.5 En pratique . . . . .	11
2.6 Exercice . . . . .	12
<b>II Structures de données</b>	<b>13</b>
<b>3 Données</b>	<b>15</b>
<b>4 Structures de données de base</b>	<b>17</b>
4.1 Containers . . . . .	17
4.2 Tables . . . . .	18
4.3 Piles . . . . .	20
4.4 Files . . . . .	22
4.5 Listes chaînées . . . . .	24
<b>5 Arbres</b>	<b>29</b>
5.1 Arbres binaires . . . . .	29
5.2 Arbres n-aires . . . . .	29
<b>6 Dictionnaires</b>	<b>31</b>
6.1 Spécification . . . . .	31
6.2 Implémentation avec une table . . . . .	32

6.3	Tables de hachage . . . . .	32
6.4	Arbres binaires de recherche . . . . .	33
6.5	Arbres binaire de recherche balancés . . . . .	33
<b>7</b>	<b>Performance pour les containers</b>	<b>35</b>
<b>8</b>	<b>Files de priorité et tas</b>	<b>37</b>
8.1	Spécification . . . . .	37
8.2	Implémentation avec une table . . . . .	38
8.3	Tas . . . . .	38
8.4	Implémentation à l'aide d'un tableau . . . . .	39
8.5	Construction d'un tas à partir d'un tableau . . . . .	40
8.6	Complexité des opérations . . . . .	41
<b>9</b>	<b>Union-find</b>	<b>43</b>
<b>10</b>	<b>Graphes</b>	<b>45</b>
10.1	Définitions . . . . .	45
10.2	Implémentation par matrice d'adjacence . . . . .	47
10.3	Implémentation par liste d'adjacence . . . . .	48
10.4	Implémentation par liste d'incidence . . . . .	48
<b>11</b>	<b>En Python</b>	<b>49</b>
11.1	Piles et files . . . . .	49
11.2	Listes et tuples Python . . . . .	50
11.3	Dictionnaires . . . . .	51
11.4	Files de priorité . . . . .	52
11.5	Ensembles . . . . .	52
11.6	Arbres binaires de recherche . . . . .	53
11.7	Graphes . . . . .	53
<b>III</b>	<b>Techniques fondamentales</b>	<b>55</b>
<b>12</b>	<b>Problèmes</b>	<b>57</b>
12.1	Définition . . . . .	57
<b>13</b>	<b>Méthode Glouton</b>	<b>59</b>
13.1	Définition . . . . .	59
13.2	Exemple du classement par insertion . . . . .	59
13.3	Exemple du rendu de monnaie . . . . .	59
13.4	Exemple de la coloration de graphe . . . . .	61
13.5	Glouton et optimalité . . . . .	63
<b>14</b>	<b>Méthode diviser pour régner</b>	<b>65</b>
14.1	Définition . . . . .	65
14.2	Algorithme de Karatsuba . . . . .	65
14.3	Analyse des algorithmes diviser pour régner . . . . .	67

14.4 Théorème maître . . . . .	68
14.5 Exemples d'application du théorème maître . . . . .	69
<b>15 Programmation dynamique</b>	<b>71</b>
15.1 Principe . . . . .	71
15.2 Exemple 1 : le rendu de monnaie . . . . .	72
15.3 Exemple 2 : le sac à dos . . . . .	73
<b>16 Exploration en espace des états</b>	<b>75</b>
16.1 Espace des états . . . . .	75
16.2 Exemples de problèmes : . . . . .	75
16.3 Performance . . . . .	77
16.4 Backtracking . . . . .	77
16.5 Backtracking : coloration de graphe . . . . .	78
16.6 Coloration de graphe : <i>forward checking</i> . . . . .	80
16.7 Exploration générique . . . . .	83
16.8 Exploration en profondeur . . . . .	84
16.9 Exploration en largeur . . . . .	84
16.10 Exploration à coût uniforme ou UCS . . . . .	84
16.11 Exploration <i>Best-First</i> . . . . .	84
16.12 Exploration A* . . . . .	85
<b>17 Méta-heuristiques</b>	<b>87</b>
17.1 Méta ... quoi? . . . . .	87
17.2 Branch and Bound . . . . .	87
17.3 Branch and bound : sac à dos . . . . .	88
<b>IV Algorithmes</b>	<b>91</b>
<b>18 Classement</b>	<b>93</b>
18.1 Définition formelle du problème du classement . . . . .	93
18.2 Classement par insertion . . . . .	93
18.3 Classement par fusion . . . . .	95
18.4 Classement rapide . . . . .	96
18.5 Classement par tas . . . . .	97
18.6 En pratique . . . . .	97
<b>19 Graphes</b>	<b>99</b>
19.1 Parcours . . . . .	99
19.2 Arbres recouvrants de poids minimal . . . . .	102
19.3 Recherche des plus courts chemins . . . . .	106
19.4 Similitudes . . . . .	111
<b>V Problèmes</b>	<b>113</b>
<b>20 Problèmes sans solution</b>	<b>115</b>

20.1 Le problème de l'arrêt . . . . .	116
20.2 Autres problèmes insolubles algorithmiquement . . . . .	117
<b>21 Problèmes faciles versus difficiles</b>	<b>119</b>
21.1 Problèmes et instances . . . . .	119
21.2 Classe P . . . . .	119
21.3 Classe NP . . . . .	120
21.4 Réduction polynomiale . . . . .	120
21.5 NP-complétude . . . . .	121
<b>22 Complexité de problème</b>	<b>123</b>
<b>VI Annexes</b>	<b>125</b>
<b>23 Spécifications</b>	<b>127</b>
<b>24 Compléments Python</b>	<b>131</b>
24.1 Lambda expressions . . . . .	131
24.2 Exceptions . . . . .	132
<b>25 Programmation objet en Python</b>	<b>135</b>
25.1 Introduction . . . . .	135
25.2 Classes et instances . . . . .	135
25.3 Méthodes . . . . .	135
25.4 Héritage . . . . .	135
<b>26 Réalisation Python de structures de données classiques</b>	<b>137</b>
26.1 Liste chaînée . . . . .	137
26.2 Arbres binaires de recherche . . . . .	138
26.3 Dictionnaires . . . . .	139
26.4 Tas . . . . .	142



# **Première partie**

## **Introduction**



## Ressources

En utilisant un ordinateur, nous sommes consommateurs de deux ressources principales :

- du temps d'exécution sur un ou plusieurs processeurs,
- de l'espace mémoire.

La conception et le choix d'un algorithme pour résoudre un problème donné vont devoir s'attacher à économiser ces ressources.

On peut s'intéresser à d'autres ressources et on peut envisager le coût du calcul sous d'autres angles. On peut par exemple s'intéresser au rapport entre le calcul et l'énergie nécessaire à celui-ci. Si vous disposez d'un joule, que pouvez-vous calculer? Nous n'aborderons pas cette question.



## Complexité des algorithmes

### 2.1 Problème et définition

De manière formelle, la complexité en temps d'un algorithme, c'est le nombre de cycles que le processeur doit exécuter en fonction des données d'entrée pour que l'algorithme termine.

En pratique, il est très difficile de déterminer **exactement** cette fonction pour deux raisons :

1. les processeurs modernes disposent de plusieurs unités de calcul qui exécutent plusieurs instructions en parallèle quand c'est possible, ce qui fait que les instructions élémentaires sont données avec un temps d'exécution qui est un intervalle et non pas un nombre fixe de cycles,
2. la deuxième est intrinsèque aux algorithmes eux-mêmes : le plus simple des algorithmes peut avoir des comportements complètement imprévisibles, et un temps d'exécution en conséquence.

Exemple : suite de Syracuse

```
function syracuse(n)
    if n = 1
        return 1
    if n % 2 = 0
        return syracuse(n/2)
    else
        return syracuse(3*n+1)
end
```

Personne n'a jamais pu prouver la terminaison de cet algorithme.

Et pour ceux qui seraient tenter de penser que la récursion y est pour quelque chose, il suffit de regarder cette autre forme du même code pour se convaincre que ça n'a rien à voir :

```
function syracuse(n)
    while n != 1
        if (n % 2 = 0)
            n = n/2
        else
            n = 3*n+1
```

```

return n
end

```

On ne peut rien faire concernant ce second point : il y a des algorithmes pour lesquels le calcul du temps qu'ils prendront en fonction des paramètres d'entrée est pratiquement impossible à déterminer. Pire : il est même impossible d'établir un mécanisme général qui prouverait que ce temps est toujours borné quand c'est le cas !

En revanche, quand cette détermination est possible, on peut prendre des mesures assez radicales pour régler le premier point et simplifier cette détermination.

## 2.2 Pseudo-code

Chaque langage de programmation introduit des constructions qui lui sont spécifiques et qui peuvent se traduire en un nombre non négligeable d'instructions machine difficiles à compter. Pour exprimer les algorithmes, nous allons donc éliminer les langages de programmation classiques, au profit d'un pseudo-code très simplifié, proche des instructions que la machine sait exécuter. Ce pseudo-code dispose des possibilités suivantes :

- l'indentation est utilisée pour indiquer la structure de bloc (comme en Python, pas de begin..end ou autres marqueurs),
- les types connus sont les booléens, les nombres entiers, les nombres flottants et les caractères
- les expressions admises sont composées de :
  - and, or, not pour les booléens
  - +, -, \*, /, >> et << (décalage à droite et à gauche) pour les entiers
  - +, -, \*, / pour les flottants
- les variables admises sont de 3 natures différentes :
  - scalaires : elles sont d'un type de base connu
  - vectorielles : ce sont des vecteurs d'éléments qui sont tous du même type connu et indicés sur un intervalle  $[0, n - 1]$ . On utilisera la notation  $A[p..q]$  pour désigner le sous-ensemble du tableau A qui commence à l'indice p et se termine à l'indice q compris.
  - composées : dans ce cas la variable possède plusieurs propriétés qui sont elles-mêmes d'un type connu. On utilisera  $A.length$  pour se référer à la propriété d'un objet. Exemple :
    - une variable  $p$  de type personne peut posséder une propriété  $age$  de type entier que l'on notera  $p.age$ .
    - une variable  $n$  de type noeud peut posséder deux propriétés de type noeud que l'on notera  $n.filsgauche$  et  $n.filstdroit$  qui désigneront d'autres noeuds.
- la conditionnelle if bool then val\_1 else val\_2
- l'instruction de boucle définie for  $i \leftarrow depart$  to  $arrivee$
- l'instruction de boucle indéfinie while
- l'appel de fonction
- les paramètres sont passés par valeur pour les types scalaires, tous les autres sont passés par référence. Une référence qui ne réfère à aucun objet a la valeur NIL (équivalent à None en Python),
- les commentaires sont introduits par #c,
- les opérateurs booléens and et or sont court-circuitants.

Ce langage pseudo-code correspond au plus petit dénominateur commun d'un grand nombre de langages de programmation réels. Il correspond également à ce qu'il est possible d'exécuter sur le modèle de la *machine à registres*, qui correspond aux ordinateurs réels. Les instructions élémentaires de ce pseudo-code seront toutes comptées pour un temps de 1 cycle machine.

Nous ne tiendrons pas compte des différents niveaux de cache mémoire. Les instructions de calcul simple seront toutes traitées de façon équivalente : pas de différence entre une addition, une multiplication ou une division.

Attention : ceci est valable pour l'arithmétique entière mais peut devenir plus complexe pour l'arithmétique en nombres flottants. Dans ce cas, une multiplication peut effectivement être bien plus couteuse qu'une addition. Il pourra arriver que nous distinguions les multiplications et divisions de nombres réels qui sont plus couteuses que les additions ou soustractions.

Attention : même en arithmétique entière, il y a des limites. L'exponentiation en est une : on ne peut pas calculer  $i^k$  en temps constant, sauf quand  $i = 2$  (Pourquoi?).

## 2.3 Complexité asymptotique

En ayant posé ce cadre, il est beaucoup plus simple à la fois d'exprimer l'essence d'une méthode algorithmique et de déterminer le temps qu'elle peut prendre en fonction de la taille de ses instances (voir définition d'un problème). Malgré tout, la détermination exacte du temps d'exécution peut encore être trop complexe.

On va donc très souvent s'intéresser aux bornes du temps d'exécution : quel est le temps minimum ou maximum pour une instance donnée, ou encore au temps moyen pour une donnée de taille fixée.

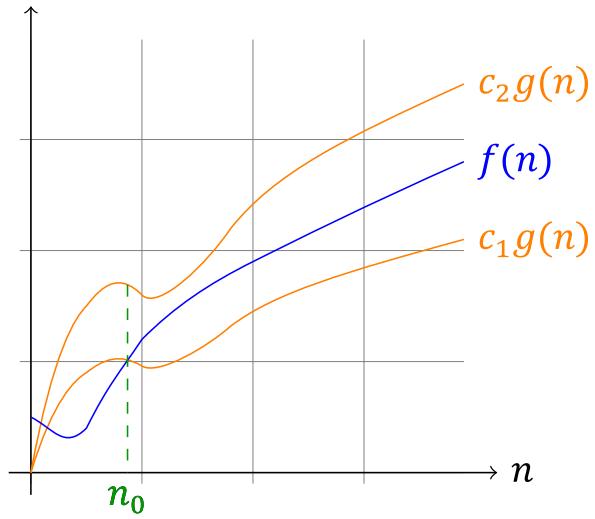
De plus, on considérera la façon dont croît ce temps de calcul en fonction de la taille des données d'entrée et on s'intéressera au *comportement asymptotique* des fonctions au voisinage de l'infini.

On va définir des notations qui vont nous permettre de quantifier le comportement d'un temps de calcul par rapport à des fonctions dont le comportement est connu. On pourra ainsi dire au choix que le temps de calcul croît *plus vite que, de façon identique à* ou *pas plus vite que* une certaine fonction comme  $n$  ou  $n^2$  ou  $2^n$ .

La notation  $\Theta()$  définit des bornes asymptotiques serrées :

$$\begin{aligned}\Theta(g(n)) = \{f(n) &| \exists c_1 > 0, c_2 > 0, n_0 > 0 \\ &\forall n > n_0 \quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}\end{aligned}$$

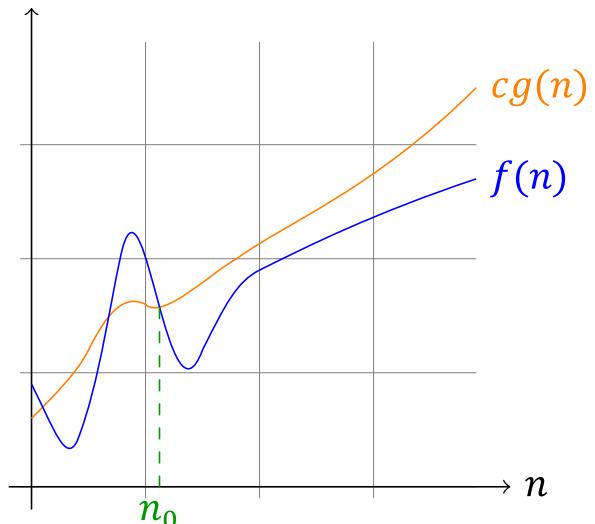
On voit donc qu'on définit une famille de fonctions dont la croissance est encadrée de façon assez stricte par la fonction  $g(n)$ .



$$f(n) = \Theta(g(n))$$

La notation  $O()$  définit une borne asymptotique supérieure :

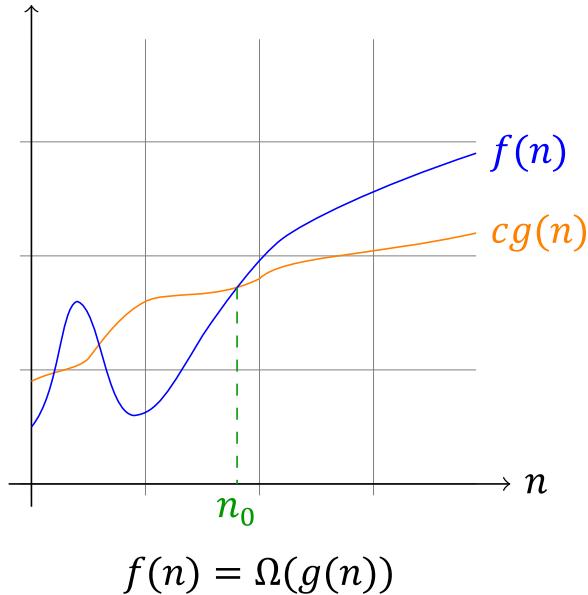
$$O(g(n)) = \{ f(n) \mid \exists c > 0, n_0 > 0 \quad \forall n > n_0 \quad 0 \leq f(n) \leq cg(n) \}$$



$$f(n) = O(g(n))$$

La notation  $\Omega()$  définit une borne asymptotique inférieure :

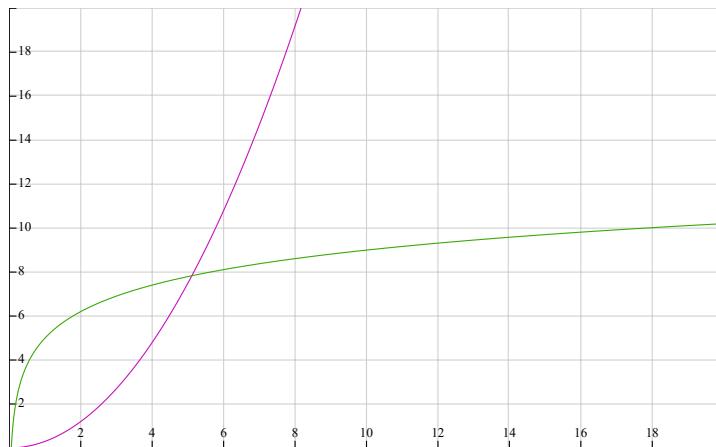
$$\Omega(g(n)) = \{ f(n) \mid \exists c > 0, n_0 > 0 \quad \forall n > n_0 \quad 0 \leq cg(n) \leq f(n) \}$$



Par abus de langage, on dira que  $f(n) = O(n^2)$  alors qu'on devrait dire en fait  $f(n) \in O(n^2)$ .

Le théorème suivant découle des définitions : pour deux fonctions  $f(n)$  et  $g(n)$ , on a la propriété  $f(n) = \Theta(g(n))$  si et seulement si  $f(n) = O(g(n))$  et  $f(n) = \Omega(g(n))$ .

Attention : il est évident sur ces schémas que ce qui est vrai *asymptotiquement* ne l'est pas obligatoirement au voisinage de zéro. En particulier, entre deux fonctions  $f_1$  et  $f_2$ , la première qui croît très rapidement et la seconde beaucoup moins, il est vraisemblable que  $f_1$  va rapidement dépasser  $f_2$ , en revanche pour de faibles valeur de  $n$ , il est possible que ce soit  $f_2$  qui domine  $f_1$ .



Le comportement asymptotique est utile et intéressant, mais il ne faut pas oublier de se demander : en pratique, quelles tailles de données vais-je traiter ? Suis-je bien dans le comportement asymptotique ?

## 2.4 Règles de calcul

On s'intéresse surtout à la notion de borne supérieure  $O()$ . Si on peut trouver une fonction qui soit un encadrement  $\Theta()$ , c'est mieux, mais ce n'est pas toujours possible. Pour beaucoup d'algorithmes, on cherchera un majorant  $O()$  du cas moyen, et également un majorant des meilleurs et pire cas. Ces trois majorants ne sont pas toujours dans la même famille de fonctions. Exemple de l'algorithme de classement rapide : le pire cas est en  $O(n^2)$  et le cas moyen en  $O(n \log n)$ .

Complexité  $O(1)$  : toute suite d'instructions en nombre constant, indépendant de la taille du problème :

```
x ← (x+y)/2
y ← 2*x - y
x ← 2*x - y
```

Complexité  $O(n)$  : cas d'une boucle exécutée un nombre de fois proportionnel à la taille du problème :

```
for i ← 1 to n
    a[i] = a[i]*a[i]
```

Complexité  $O(n^2)$  : cas de deux boucles **imbriquées** exécutées **chacune** un nombre de fois proportionnel à la taille du problème :

```
function search(A[])
    found = false
    for i ← 0 to n-1
        for j ← 0 to n-1
            if i ≠ j ∧ A[i] = A[j]
                found = true
    return found
```

Complexité  $O(\sqrt{n})$  : cas d'une boucle qu'on exécute en nombre proportionnel à la racine carrée de la taille du problème :

```
function test(n)
    i ← 2
    r ← true
    while i*i < n
        if n % i = 0
            r ← false
    return r
```

Complexité  $O(\log n)$  : cas d'une boucle où l'on divise  $n$  par 2 à chaque tour :

```
function search(A, k, a, b)
    while b > a
        c ← (a+b)/2
        if A[c] = k
            return c
        if A[c] > k
```

```

b ← c
else
    a ← c
return -1

```

Les règles de calcul avec la notation  $O()$  sont assez immédiates.

**Théorème 2.1.** *Pour toute valeur constante  $c$ ,  $cf(n) \in O(f(n))$ .*

On en déduit que  $O(cf(n)) = O(f(n))$ . Les constantes sont absorbées dans la famille de fonctions.

**Théorème 2.2.** *Transitivité. Si  $f(n) \in O(g(n))$  et  $g(n) \in O(h(n))$  alors  $f(n) \in O(h(n))$ .*

**Théorème 2.3.** *Addition. Si  $g_1(n) \in O(f_1(n))$  et  $g_2(n) \in O(f_2(n))$ , alors  $g_1(n) + g_2(n) \in O(\max f_1(n), f_2(n))$ .*

**Théorème 2.4.** *Produit. Si  $g_1(n) \in O(f_1(n))$  et  $g_2(n) \in O(f_2(n))$ , alors  $g_1(n)g_2(n) \in O(f_1(n)f_2(n))$ .*

**Théorème 2.5.** *Limites. Supposons que la limite  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L$  existe. Dans ce cas :*

- si  $L = 0$ , alors  $f(n) \in O(g(n))$
- si  $0 < L < +\infty$ , alors  $f(n) \in \Theta(g(n))$
- si  $L = +\infty$ , alors  $f(n) \in \Omega(g(n))$ .

## 2.5 En pratique

Voici les fonctions usuelles que l'on rencontre, par ordre de croissance :

Complexité	nom
$O(1)$	constante
$O(\log n)$	logarithmique
$O((\log n)^c)$	polylogarithmique
$O(n)$	linéaire
$O(n \log n)$	
$O(n^2)$	quadratique
$O(n^c)$	polynomiale
$O(c^n)$	exponentielle

De façon moins usuelle, on peut également rencontrer des termes  $\log \log n$ .

Il n'y a aucun algorithme « pratique », i.e. polynomial, avec un degré supérieur à 5.

On peut avoir besoin d'exprimer une complexité en fonction de deux paramètres  $n$  et  $m$  qui représentent deux dimensions de la taille du problème d'entrée. Ce sera notamment le cas pour les problèmes de graphes, dont la dimension est caractérisée par le nombre de sommets du graphe et le nombre d'arêtes du graphe.

## 2.6 Exercice

Quelle est la complexité de l'algorithme suivant en fonction de la donnée d'entrée  $n$  :

```
function calcul(n)
    nb ← 0
    i ← 2
    while i ≤ n*n*n
        nb ← nb+1
        i ← i*2
    return nb
```

## **Deuxième partie**

# **Structures de données**



## Données

Les *structures de données* ont pour objet d'organiser les données de nos problèmes de la façon la plus efficace possible en vue de leur traitement.

On peut regarder les structures de données selon plusieurs angles : depuis le plus abstrait – les opérations qu'elles prennent en charge – jusqu'au plus pratique – la façon dont elles sont implémentées.

Nous allons nous intéresser essentiellement à deux types de structures de données : l'une pour stocker des objets que l'on nommera *container*, l'autre pour représenter des *graphes*. Les containers et les graphes ont une utilisation très générale et sont omniprésents en informatique. D'autres structures de données plus spécifiques ont été étudiées pour d'autres besoins, comme par exemple les chaînes de caractères (séquencement de génome, etc.) ou les formes géométriques (réalité virtuelle, véhicules autonomes, etc.).

On peut réaliser des containers de bien des manières différentes, et cette variété de réalisations possibles est essentiellement dictée par des préoccupations de performance des opérations mais également des préoccupations d'utilisation de la mémoire.

L'ordinateur fournit nativement une structure de donnée de base, de manière physique : la mémoire de l'ordinateur peut être vue comme un tableau dont les adresses varient de 0 à  $2^n - 1$ .

Pour utiliser cette mémoire, le système d'exploitation doit nous fournir *à la demande* des blocs de mémoire de taille donnée. Nous introduisons donc la notion de mécanisme d'allocation mémoire.

Sous sa forme la plus simple, le processus d'allocation mémoire est réalisé par deux opérations :

1. demande d'allocation d'un bloc de taille  $n$
2. libération d'un bloc précédemment alloué.

Les blocs sont de taille fixe et une fois un bloc alloué, sa taille n'est plus modifiable. Si on veut un bloc plus grand, il faut en allouer un nouveau et recopier les données de l'ancien bloc dans le nouveau bloc. La libération est prise en charge de façon automatique par la plupart des langages de programmation aujourd'hui (exceptions notables : C/C++). C'est le *ramasse-miettes* ou *garbage collector* qui se charge de cette tâche.

Lors du calcul de complexité en temps d'un algorithme, on compte l'allocation mémoire pour un temps constant  $O(1)$ . C'est une approximation en général correcte. Toutefois, le processus d'allocation mémoire induit une fragmentation de la mémoire qui doit être prise en compte et traitée. Potentiellement, on s'expose à un ralentissement important si l'on fragmente trop la mémoire. En

effet, la recherche d'un bloc de taille adéquate dans une mémoire fragmentée peut nécessiter la réorganisation des blocs mémoire libres et ce processus peut-prendre un temps non négligeable, et surtout non constant, mais dépendant du nombre d'allocations ayant eu lieu auparavant. Cette réorganisation peut aussi être prise en charge par le ramasse-miettes. Dans tous les cas, il faut garder à l'esprit que l'utilisation de la mémoire ne doit pas se faire de façon anarchique mais doit rester la plus rationnelle possible.

# Structures de données de base

## 4.1 Containers

### Spécification

On peut voir les containers selon trois angles :

- stockage : ils permettent de mémoriser une information
- accès : ils permettent de retrouver une information mémorisée.
- parcours ou balayage : ils permettent de passer en revue tous les éléments qu'ils contiennent.

Nous allons donner une spécification des containers d'un point de vue opérationnel. Cette spécification *n'est pas du code* : elle a pour rôle de définir de manière abstraite les opérations sur les données, ainsi que d'exiger des propriétés sur ces opérations. Voir le lien et en annexe pour plus de précisions sur la nature de cette spécification.

**Spec Container**

**Operations :**

newContainer	:	→ Container
isEmpty	:	Container → Bool
insert	:	Container Element → Container
search	:	Container Key → Element
delete	:	Container Element → Container
minimum	:	Container → Element
maximum	:	Container → Element
successor	:	Container Element → Element
predecessor	:	Container Element → Element

**Preconditions :**

successor(C, e) ⇒ e ≠ maximum(C)
predecessor(C, e) ⇒ e ≠ minimum(C)

**Axioms :**

**EndSpec**

Le container est une abstraction qu'il faudra réaliser concrètement à l'aide de structures de données spécifiques. Différentes réalisations induiront des performances différentes.

La spécification permet d'écrire les *profils* (ou *signature*) d'opérations que le container doit fournir. On peut également donner des *préconditions* à certaines opérations : ici, on ne peut pas parler du successeur de l'élément maximal. On verra des exemples d'*axiomes* qui permettent de donner des conditions fortes sur la nature du traitement réalisé par une opération.

Le parcours des éléments d'un container est couramment associé avec la notion d'*itérateur*. Ces itérateurs prennent diverses formes selon les langages.

### **Allocation statique versus dynamique**

Les containers peuvent arriver dans deux versions selon les langages de programmation et l'implémentation de la structure de donnée. Ils peuvent être de taille fixe ou bien extensibles. Ceci ne se voit pas nécessairement dans la spécification.

La différence essentielle entre les deux approches tient à la performance : les structures de données extensibles sont en général plus lentes que les structures de données allouées statiquement. En revanche, elles utilisent la mémoire de façon plus parcimonieuse.

Une structure de donnée statique verra sa mémoire allouée soit au chargement du programme, soit lors d'une seule allocation initiale à l'exécution. La mémoire allouée dans ce cas aura de grandes chances d'être constituée d'un seul bloc contigu. Si de plus ce bloc est parcouru de façon linéaire, on optimise encore l'utilisation du matériel (voir un cours d'architecture des systèmes informatiques).

Une structure de donnée dynamique allouera de façon répétée et à la demande, des petits blocs de mémoire. Éventuellement, un grand nombre de ces blocs auront été alloués. Ces blocs étant a priori indépendants les uns des autres, il faudra prévoir de les *lier* par des *pointeurs* : il faut qu'un bloc initial connaisse un ou des suivants. Il y a donc une perte de place induite par la nécessité de lier les blocs entre eux. D'autre part, l'allocation et la libération aléatoire de blocs mémoire génère une fragmentation qui va engendrer également une perte de performance. En revanche, une structure de donnée dynamique autorisera la *libération* de blocs mémoire inutilisés, ce qu'une structure de donnée statique ne pourra pas faire.

En pratique, une première règle consiste à se demander : est-ce que je peux prévoir à l'avance combien de mémoire sera utilisée par mon programme en fonction des données initiales du problème ? Si oui, il est préférable d'utiliser une structure statique, car elle sera plus efficace. Dans le cas contraire, seule une structure dynamique répondra au besoin d'extensibilité sans avoir à allouer systématiquement une taille fixe maximale qui ne serait pas utilisée.

## **4.2 Tables**

### **Spécification**

Une table est une structure de donnée très simple : elle supporte uniquement deux opérations *put* et *get* pour insérer et retrouver des objets dans la table. Chaque objet est indexé : l'endroit où il est stocké dans la table est déterminé par un indice. Dans le cas le plus simple, l'ensemble des indices est un intervalle du type  $[0, n - 1]$  pour une table à  $n$  éléments, mais l'ensemble des indices peut également être un  $n$ -uplet dans le cas d'une matrice à plusieurs dimensions. Nous verrons plus loin que l'on peut-même envisager des tables avec des ensembles d'indices quelconques.

Attention : la table ne change jamais de taille. Lorsqu'on veut enlever un objet, il faut simplement vider la case où il se trouvait.

Attention : les tables sont toujours passées par référence lors d'un appel de fonction. Si vous modifiez la table dans la fonction appelée, elle revient modifiée dans la fonction appelante. Ceci tient à des raisons de performance : une table est potentiellement large, on ne la recopie donc que lorsque le programmeur le demande explicitement.

```
Spec Array

Operations :
  get : Array Int → Element
  set : Array Int Element → Array

Axioms :
  a    : Array
  i, j : Int
  e    : Element

  get(set(a, i, e), i) = e
  i ≠ j ⇒ get(set(a, i, e), j) = get(a, j)

EndSpec
```

On voit ici un exemple d'*axiomes* qui exigent que lorsqu'on met un objet à un indice donné dans la table, on le retrouve bien ultérieurement au même endroit.

## Implémentation

La plupart des langages de programmation fournissent nativement des tableaux indexés par des nombres entiers. Aujourd'hui la majorité des langages utilisent des indices comptés à partir de 0 et c'est ce que nous ferons également dans notre pseudo-code.

La traduction des opérations `get` et `set` dans un langage de programmation classique est immédiate : `a[i]` pour `get(a,i)` et `a[i] = e` pour `set(a,i,e)`.

Pour remédier au problème de l'extensibilité, une pratique couramment adoptée consiste à détecter le débordement de capacité du tableau initial, et à allouer un nouveau bloc plus grand (en général de taille double) dans lequel on recopie toutes les informations du bloc précédent. On libère ensuite le bloc initial et ainsi de suite. Cette solution est effective, converge assez rapidement vers une situation où la table n'a plus besoin d'être agrandie ... au risque d'allouer quasiment le double de la mémoire nécessaire. La détection du débordement de capacité est possible car l'accès dans la table à un indice trop grand déclenche une exception (voir exceptions en Python) du type `ArrayOutOfBoundsException` (ou similaire). Cette exception peut être traitée par la réallocation de la table et le programme pourra continuer.

La plupart des langages fournissent également des tableaux multidimensionnels indexés dans  $\mathbb{N} \times \mathbb{N} \times \dots$ .

## Performance

Les opérations `get` et `put` sont en temps constant  $O(1)$ .

Pour réaliser un container sur la base d'une table, deux options s'offrent à nous :

1. on insère chaque élément à la prochaine place vide en partant du début du tableau. Dans ce cas, les opérations `insert` et `search` sont en temps linéaire  $O(n)$ .
2. on maintient un ordre sur les éléments du tableau lors de l'insertion d'un nouvel élément. Dans ce cas, l'opération `insert` se déroule en temps linéaire  $O(n)$  mais l'opération `search` se déroulera en temps logarithmique  $O(\log n)$  car on pourra effectuer une recherche dichotomique.

### 4.3 Piles

#### Spécification

Une *pile* est une structure de donnée de type container qui limite l'accès à ses éléments : on ne peut accéder aux éléments que dans l'ordre *dernier entré, premier sorti* ou *last-in, first-out* ou LIFO.

```

Spec Stack

Operations:
newStack :           → Stack
isEmpty   : Stack    → Bool
push      : Stack Element → Stack
pop       : Stack    → Stack
top       : Stack    → Element

Axioms:
s : Stack
e : Element

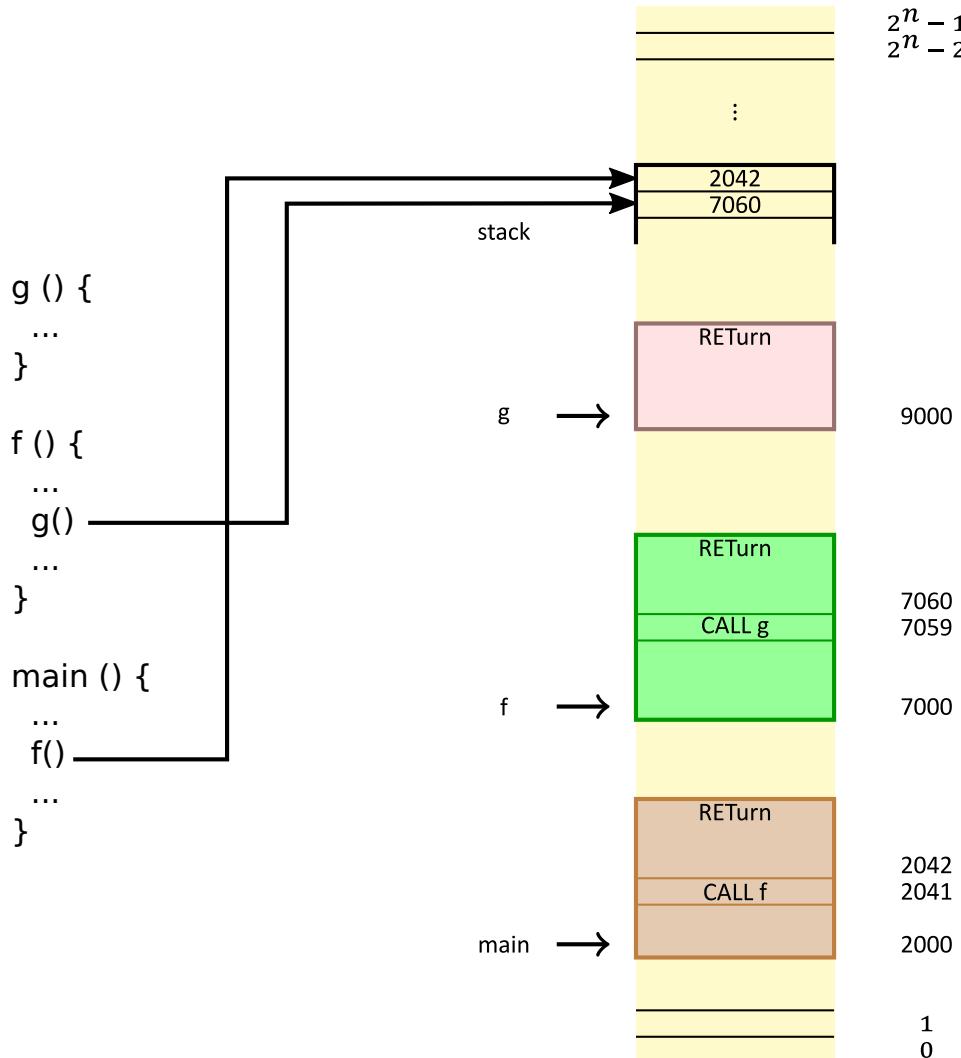
pop(push(s, e)) = s
top(push(s, e)) = e
isEmpty(emptyStack) = true
isEmpty(push(s, e)) = false

EndSpec

```

Le fonctionnement est extrêmement simple : on peut empiler des éléments. Si l'on veut retrouver le troisième élément dans la pile, il faut dépiler (et perdre, ou mémoriser ailleurs) les deux premiers.

Ce mécanisme est directement nécessaire dans l'ordinateur lui-même pour réaliser les appels de fonction. Sans lui, ni les appels récursifs de fonctions, ni même les appels imbriqués ne seraient possibles.



On a ici une illustration d'une séquence d'appels de fonctions : la fonction `main()` appelle la fonction `f()` qui appelle la fonction `g()`. Pour que ce mécanisme soit générique, à chaque appel de fonction, il faut *empiler* l'adresse à laquelle il faudra revenir lorsque l'exécution de la fonction sera terminée. On voit aussi clairement que seul un mécanisme de pile autorise des appels récursifs. En général, le nombre d'appels d'une fonction récursive dépend des paramètres de la fonction. La pile possède un autre rôle : on passe les arguments de la fonction par la pile également. Il faudra donc une pile suffisamment grande pour enregistrer toutes les informations nécessaires lors d'une exécution du programme, sous peine de recevoir un message d'erreur du type *Stack Overflow* (débordement de pile).

### Implémentation à l'aide d'un tableau

Si on utilise un tableau de taille  $N$ , on pourra seulement stocker  $N$  éléments dans la pile. Au delà, on déclenchera une *exception* de type *stack overflow* (débordement de pile).

```
function newStack()
  #c Allouer un tableau assez grand
```

```

#c pour contenir un nombre maximum
#c d'éléments
p.iTop ← -1
end

function isEmpty(p)
    return p.iTop = -1
end

function top(p)
    if isEmpty(p)
        Error("Empty stack")
    return p[p.iTop]
end

function push(p, e)
    p.iTop ← p.iTop+1
    if p.iTop = p.length
        Error("Pile pleine")
    p[p.iTop] ← e
    return p
end

function pop(p)
    if (isEmpty(p))
        Error("Pile vide")
    p.iTop ← p.iTop-1
    return p
end

```

## Performance

Toutes les opérations se déroulent en temps constant  $O(1)$ , que l'on implémente la pile avec un tableau ou avec une liste chaînée.

## 4.4 Files

### Spécification

Une *file* est une structure de donnée de type container qui limite l'accès à ses éléments : on ne peut accéder aux éléments que dans un ordre *premier entré, premier sorti* ou *first-in, first-out* ou FIFO.

Spec Queue

Operations :

```

newQueue : → Queue
isEmpty    : Queue → Bool
enqueue   : Queue Element → Queue
dequeue   : Queue → Queue
first     : Queue → Element

Preconditions :
dequeue(q) ⇒ emptyQueue(q) = false
first(q)   ⇒ emptyQueue(q) = false

Axioms :
q : Queue
e, e1, e2 : Element

dequeue(enqueue(emptyQueue,e)) = emptyQueue
dequeue(enqueue(enqueue(q,e1),e2)) = enqueue(dequeue(enqueue(q,e1)),e2)
first(enqueue(emptyQueue,e)) = e
first(enqueue(enqueue(q,e1),e2)) = first(enqueue(q,e1))
isEmpty(emptyQueue) = true
isEmpty(enqueue(q,e)) = false

EndSpec

```

Les éléments sont insérés à une extrémité de la file et sortis à l'autre extrémité. On ne peut pas changer leur ordre ni accéder aux éléments qui sont à l'intérieur de la file.

On utilise ce mécanisme y compris dans des dispositifs matériels (*hardware*) lorsqu'il faut faire communiquer deux dispositifs asynchrones : l'émetteur et le récepteur fonctionnent sans synchronisation. Dans ce cas, un tampon FIFO entre les deux permet de palier une éventuelle émission trop rapide (au moins pendant un certain temps).

### Implémentation à l'aide d'un tableau

Avec un tableau de taille fixe, il faut ruser pour utiliser la capacité totale du tableau pour stocker des éléments. En effet, selon l'ordre des opérations enqueue et dequeue, nous pouvons assez rapidement arriver à une situation où nous ne pouvons plus insérer d'éléments alors qu'il reste de la place dans le tableau.

Nous pouvons remédier à ce problème en rendant notre tableau (ou *buffer*) *circulaire* : il suffit de calculer les indices modulo la taille du tableau et de faire attention à ce qu'ils ne se croisent pas.

```

function newQueue()
  f.iHead ← 0
  f.iTail ← f.length - 1
  f.count ← 0
end

function first(f)

```

```

if f.count = 0
    Error("File vide")
return f[iHead]
end

function enqueue(f, e)
    if f.count = f.length
        Error("File pleine")
    f[iTail] ← e
    f.count ← f.count + 1
    f.iTail = (f.iTail + 1) % f.length
    return f
end

function dequeue(f)
    if f.count = 0
        Error("File vide")
    f.count ← f.count - 1
    f.iHead = (f.iHead + 1) % f.length
    return f
end

#c Pour tenir compte de la taille de la file
#c et détecter les conditions file pleine
#c et file vide, il faut tenir un compte
#c du nombre d'éléments présents dans la file

```

## Performance

Toutes les opérations se déroulent en temps constant  $O(1)$ , que l'on implémente la pile avec un tableau ou avec une liste chaînée.

## 4.5 Listes chaînées

### Spécification

Il existe beaucoup d'interprétations du mot *liste* en informatique, ce qui en fait un mot assez vague. En particulier les `list` de Python et les `ArrayList` de Java sont des structures très différentes de ce que nous appellerons ici *liste chaînée*.

Nous décrivons ici une version très classique de la *liste chaînée* obtenue par *chaînage* linéaire d'éléments indépendants appelés *noeuds*.

C'est un premier exemple de structure de donnée dynamique : son objectif est de permettre d'ajouter et de retirer dynamiquement des noeuds de façon efficace.

**Spec List**

```

Operations :
 newList : → List
 isEmpty  : List → Bool
 length   : List → Int
 insert   : List Element → List
 first    : List → Element
 rest     : List → List
 delete   : List Element → List
 search   : List Element → Element
 getAt    : List Int → Element
 insertAt : List Element Int → Element
 deleteAt : List Int → List

Preconditions :
 first(l) ⇒ ¬ isEmpty(l)
 insertAt(l, e, i) ⇒ i ≥ 0 ∧ i ≤ length(l)
 deleteAt(l, i) ⇒ i ≥ 0 ∧ i < length(l)

Axioms :
 isEmpty(emptyList) = true
 isEmpty(insert(l, e)) = false
 first(insert(l, e)) = e
 rest(emptyList) = emptyList
 rest(insert(l, e)) = l
 ...
EndSpec

```

On peut implémenter les opérations des containers au-dessus de la liste chaînée.

## Implémentation

L'implémentation d'une liste chaînée dynamique repose sur le capacité du système / du langage de programmation à allouer et libérer dynamiquement, à la demande, des *maillons* de la liste. Nous allons ici donner le pseudo-code pour une liste doublement chaînée. L'accès à l'élément précédent un élément donné est en effet souvent nécessaire.

L'opération `newList()` consiste essentiellement à initialiser la liste de façon à ce qu'elle soit vide.

```

function newList()
 l ← nouvelle liste
 l.head ← nil
 return l

```

L'insertion d'un nouvel élément a lieu en tête de liste :

```

function insert(L, e)
 x.next ← L.head
 x.prev ← nil

```

```

if L.head ≠ nil
    L.head.prev ← x
    L.head ← x
return L

```

L'opération `insert()` a une complexité en  $O(1)$ .

La recherche d'un élément de clé  $k$  donnée s'effectue séquentiellement :

```

function search(L, k)
    x ← L.head
    while x ≠ nil ∧ x.key ≠ k
        x ← x.next
    return x

```

L'opération `search()` a une complexité en  $O(n)$  si la liste compte  $n$  éléments.

La suppression d'un élément **connu** (on n'a pas à le chercher !) s'effectue immédiatement avec une complexité en  $O(1)$  :

```

function delete(L, x)
    if x.prev ≠ nil
        x.prev.next ← x.next
    else
        L.head ← x.next
    if x.next ≠ nil
        x.next.prev ← x.prev
    return L

```

Les opérations d'insertion et de suppression sont illustrées ci-dessous.

## Performance

Les opérations d'insertion et de suppression se déroulent en temps constant  $O(1)$ . Il faut toutefois préciser ce qu'on entend par insertion et suppression : il ne s'agit pas ici d'accéder à un élément de clé donnée. L'insertion dans la liste peut s'effectuer en tête de liste pour une coût constant. Si la liste est doublement chaînée, l'insertion peut aussi avoir lieu à la fin de la liste.

La suppression suppose qu'on connaisse l'élément à supprimer, ou plutôt son prédécesseur dans la liste chaînée. Dans ce cas, on peut rerouter les pointeurs e façon à court-circuiter l'élément à supprimer de la liste pour un temps constant. Attention : dans une liste simplement chaînée, si on a seulement l'élément à supprimer, il faut rerouter les pointeurs à partir de son prédécesseur et l'obtention du prédécesseur se fait en  $O(n)$ , car il faut repartir du début de la liste chaînée pour trouver le prédécesseur de l'élément courant.

L'opération d'accès à un élément de rang donné en revanche, de par la nature même de la liste chaînée, se déroule en temps linéaire  $O(n)$  : pour accéder à l'élément de rang  $i$ , il faut parcourir tous ceux de rang compris entre 0 et  $i - 1$ . De ce fait, même si on maintient les éléments en ordre lors de leur insertion, il est inutile d'effectuer une recherche dichotomique dans une liste chaînée parce qu'on ne possède pas d'opération d'accès direct à un élément en temps constant. En effet la recherche dichotomique suppose que l'on sache accéder à un élément d'indice quelconque dans la liste. L'équation de récurrence temporelle pour la recherche dichotomique est  $T(n) = T(\frac{n}{2}) +$

$f(n)$  : à partir d'un intervalle de taille  $n$ , on choisit l'un des deux intervalles de taille  $\frac{n}{2}$  dans lequel se situe notre solution. La fonction  $f(n)$  correspond au temps mis par l'opération de choix de cet intervalle. Si l'accès à un élément d'indice donné est en  $O(1)$ , alors  $f(n) = c$ , mais si l'accès est en  $O(n)$ , alors  $f(n) = cn + d$ . Dans le premier cas (celui des tableaux), la dichotomie donne  $O(\log n)$  mais dans le second cas, elle donne  $O(n \log n)$ . Une recherche dichotomique dans une liste chaînée est donc moins performante qu'une recherche directe, linéaire, dans la liste chaînée. On pouvait l'anticiper : la recherche dichotomique demande fréquemment de repartir de la tête de la liste pour accéder à un élément d'indice donné. Au total, on parcourt beaucoup trop souvent la liste.

On peut bien entendu utiliser des listes pour implémenter des piles et des files qui ne souffrent pas d'une capacité limitée. On pourra facilement implémenter une pile avec une liste simplement chaînée où l'on empile et dépile les éléments en tête de liste. Une file en revanche demandera que l'on garde une référence sur la tête de la liste et une autre sur la fin de la liste. Il faudra également utiliser une liste doublement chaînée pour pouvoir extraire les éléments à la fin de la liste.

En termes d'espace mémoire, le chaînage occupe une place qui ne sert pas directement à stocker de l'information. On peut se poser la question de la place relative occupée par les informations de chaînage par rapport à l'information utile d'un noeud. Le chaînage double va typiquement occuper deux mots mémoire : est-ce que cette quantité est importante au regard de l'information stockée dans un noeud, ou bien est-ce négligeable ?



# Arbres

## 5.1 Arbres binaires

Nous nous sommes intéressés à des structures qui sont essentiellement linéaires : tables et listes chaînées. Nous pouvons également définir des structures bi-dimensionnelles : arbres et graphes. La notion d'arbre en tant que structure intervient naturellement : en taxonomie, en généalogie, etc. Nous allons abstraire cette structure et définir des opérations dessus.

Les arbres binaires nous serviront à deux fins :

1. la définition de *tas* pour réaliser des files de priorité,
2. la définition d'*arbres binaires de recherche* pour réaliser des *containers*.

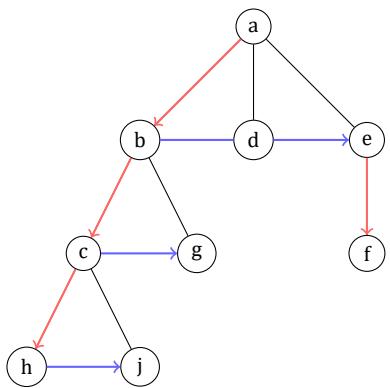
Un arbre binaire est une structure de données dynamique telle que chaque noeud possède au plus deux *fils* que l'on qualifie souvent de *fils gauche* et *fils droit*. Un noeud qui possède un noeud fils et le noeud *parent* de ce noeud fils. Un noeud parent est relié à un noeud fils par une *arête*. Chaque noeud possède au plus un parent. Un noeud qui possède au moins un fils est un noeud *interne*. Un noeud qui ne possède pas de fils est un noeud *externe*, *terminal* ou plus communément une *feuille* de l'arbre. Un arbre peut être *enraciné* ou *libre* selon qu'il possède ou non un noeud particulier désigné comme *racine* de l'arbre. Dans un arbre libre, tout noeud peut être utilisé comme racine et les liens ne sont pas orientés. Si on désigne une racine, les liens deviennent orientés, la racine étant l'ancêtre unique de tous les autres noeuds. Tout ensemble d'arbres non connectés est appelé *forêt*.

La *profondeur* d'un noeud est la distance en nombre d'arêtes entre ce noeud et la racine. La *profondeur de l'arbre* est la profondeur d'un noeud de profondeur maximale.

Un arbre binaire de profondeur  $d$  possède au plus  $1 + 2 + 2^2 + \dots + 2^{d-1} = 2^d - 1$  noeuds.

## 5.2 Arbres n-aires

On peut aisément représenter un arbre n-aire par un arbre binaire en utilisant le mécanisme décrit ci-dessous :



Les liens *fils gauche* indiquent la descendance tandis que les liens *fils droit* permettent de parcourir la liste des frères.

# Dictionnaires

## 6.1 Spécification

Un dictionnaire est une structure de donnée de type container qui va se focaliser sur l'efficacité des opérations `search()`, `insert()` et `delete()`. Là où naturellement les éléments d'une table sont indicés par des entiers naturels (ou des tuples d'entiers naturels pour une table à plusieurs dimensions), les éléments d'un dictionnaire sont indicés par les éléments de l'ensemble des clés.

Ainsi, on peut voir le dictionnaire comme une extension de la table à des ensembles d'indices quelconques. Le dictionnaire réalise une association (au sens fonctionnel) entre les éléments d'un ensemble de *clés* et les éléments d'un ensemble de *valeurs*.

La spécification générale est similaire à celle d'une table :

```
Spec Dict

Operations :
get    : Dict Key → Value
set    : Dict Key Value → Dict
delete : Dict Key → Dict

Axioms :
a      : Dict
k1, k2 : Key
v      : Value

get(set(a, k, v), k) = e
k1 ≠ k2 ⇒ get(set(a, k1, v), k2) = get(a, k2)

EndSpec
```

En pratique, on ajoute également une opération `delete()` qui a pour effet de supprimer un élément de clé donnée du dictionnaire. Cette opération n'existe pas sous la même forme avec les tableaux, puisque ceux-ci sont indicés par des entiers naturels et qu'on ne va pas supprimer un entier naturel de  $\mathbb{N}$ .

## 6.2 Implémentation avec une table

Il est bien sûr possible d'implémenter un dictionnaire de manière naïve avec une table qui enregistre les couples  $(k, v)$  de clés - valeurs .

Si la liste comporte  $n$  éléments, l'opération `get()` se déroulera en  $O(n)$ , puisqu'il faudra chercher le couple éventuel qui possède la bonne clé. Idem pour l'opération `set()`, puisqu'avant d'ajouter un couple à la table, il faut vérifier qu'il n'existe pas déjà un couple avec cette clé dans la table.

On voit que cette implémentation, même si elle est correcte, n'est pas très efficace.

Le même raisonnement vaut pour une liste chaînée à la place d'une table.

## 6.3 Tables de hachage

Dans notre dictionnaire, l'opération importante est `get` : comment accéder le plus rapidement possible à un élément de clé donnée ? Les tables nous donnent un *accès direct* aux éléments : quand on a l'indice, on peut se rendre directement à la case où doit se trouver l'élément.

Dans le cas du dictionnaire, soit  $S = ((k_1, v_1), (k_2, v_2), \dots (k_n, v_n))$  l'ensemble des éléments à enregistrer. Appelons  $U$  l'univers de toutes les clés possibles et énumérons les. C'est possible puisque chaque clé est de taille finie et l'alphabet des symboles qui permet d'écrire une clé est fini également. Nous avons donc une bijection entre  $U$  et un certain intervalle  $[0..N - 1]$  (où  $N = |U|$ ). Nous pouvons réaliser notre dictionnaire avec une table à accès direct  $A[0..N - 1]$ . Il suffit de stocker chaque élément  $(k, v)$  à  $A[k]$ .

Là où le bâton blesse, c'est que  $|U|$  peut être gigantesque, et en particulier  $|U| \gg |S|$ . Non seulement on perd énormément de place, mais ce peut être complètement impraticable :  $|U|$  peut prendre des valeurs astronomiques. Il suffit de prendre pour exemple l'ensemble des numéros de sécurité sociale versus la taille de la population française. La clé est un nombre décimal à 13 chiffres : il y a donc potentiellement  $10^{13}$  clés et seulement 70 millions d'individus. (On peut regarder la formation de la clé et s'apercevoir qu'il en existe un peu moins de valides).

Cette solution semble donc vaine au premier abord. Toutefois, nous pouvons retomber sur nos pieds avec un peu de ruse. Il suffit de définir une *injection* de  $U$  dans un ensemble de taille approximativement équivalente à celle de  $S$ .

Chaque clé  $k \in U$  est une chaîne de caractères  $k_0 k_1 \dots k_{|k|-1}$  de caractères  $k_i$  pris dans un alphabet de taille  $\alpha$ . Cet alphabet peut-être l'alphabet ASCII ou ISO Latin 1 ou etc. Calculons alors :

$$H(k) = \sum_{i=0}^{|k|-1} \alpha^{|S|-(i+1)} \times \text{char}(k_i)$$

Cette fonction  $H$  associe à chaque clé  $k$  un nombre unique en considérant les caractères de  $k$  comme des chiffres d'un nombre écrit en base  $\alpha$ . Il ne reste plus qu'à ramener ce nombre potentiellement très grand dans un intervalle  $[0..m - 1]$  en calculant  $H(k) \bmod m$ . Nous appelons cette valeur  $H(k) \bmod m$  la valeur de *hachage* de  $k$ .

Si  $m$  est "bien" choisi, les valeurs calculées seront distribuées uniformément dans l'intervalle  $[0..m - 1]$ . Ici, "bien" peut se traduire par "choisir un nombre premier pas trop proche d'une puissance de 2". La raison de ce type de choix est bien trop longue pour tenir ici. Pour une analyse détaillée du problème, se reporter à [Knuth](#).

Nous avons donc trouvé un moyen de ramener nos clés dans un ensemble de taille raisonnable. Ceci n'empêche pas que nous avons défini une injection de  $U$  dans  $[0..m - 1]$  et que potentiellement

ment, deux éléments de clés différentes auront la même valeur de hachage et entreront donc en *collision* dans la table. Comment résoudre ce nouveau problème ?

Une solution simple consiste à mettre en place un mécanisme d'*adressage ouvert*. Ce mécanisme consiste à considérer que plusieurs éléments peuvent trouver leur place au même indice dans la table. Pour celà, il faut que chaque case de la table pointe vers une liste chaînée d'éléments et non pas un seul élément.

Avec l'adressage ouvert, l'opération `set` se déroule en temps constant : il suffit de calculer l'adresse de la liste chaînée et d'insérer l'élément en tête de la liste. Les opérations `delete` et `get` nécessitent une recherche dans une liste de longueur moyenne  $\alpha = \frac{n}{m}$  qui est le taux de remplissage de la table (nombre d'éléments stockés sur nombre de places disponibles), en supposant que les clés soient uniformément distribuées que la fonction de hachage les distribue uniformément. Ces opérations ont donc une performance en  $O(1 + \alpha)$ .

Les tables de hachage sont très efficaces pour peu qu'on ne les remplisse pas de trop. Il est bien évident que si toutes les valeurs de clés sont hachées à la même place, votre valeur de  $\alpha$  va tendre vers  $n$  ! En pratique, les choses se passent souvent très bien.

Attention : les tables de hachage n'ont pas vocation à remplacer des tableaux. Si vos éléments sont naturellement indexés par des entiers et que vous n'avez pas de trous dans l'ensemble des indices, il faut utiliser des tableaux. La performance sera supérieure.

## 6.4 Arbres binaires de recherche

Pour remédier aux inconvénients des autres structures proposées jusqu'ici : l'insertion/suppression dans une table est en  $O(n)$  alors que c'est l'accès qui est en  $O(n)$  dans une liste, on peut essayer de s'appuyer sur une structure qui ne soit plus linéaire, mais à deux dimensions : un arbre.

On suppose qu'il existe une relation d'ordre entre les éléments que l'on souhaite stocker dans l'arbre. Nous allons supposer que les noeuds de l'arbre disposent d'un attribut spécifique nommé *clé* et qu'il existe une relation d'ordre entre ses clés. Un arbre binaire de recherche est un arbre binaire doté d'une contrainte sur l'ordre dans lequel les éléments peuvent apparaître dans l'arbre.

Un arbre binaire est un ABR (*Arbre Binaire de Recherche*) si et seulement si :

- la valeur de la clé à la racine est supérieure ou égale à toutes les valeurs des clés de son sous-arbre gauche
- la valeur de la clé à la racine est inférieure à toutes les valeurs des clés de son sous-arbre droit
- les sous-arbres gauche et droit sont eux-mêmes des arbres binaires de recherche.

L'idée est d'exploiter cette relation d'ordre lorsqu'on stocke des éléments dans un ABR pour accélérer sa recherche ensuite. On voit assez facilement que pour trouver un élément de clé donnée dans un ABR, il suffit de parcourir une branche de l'ABR en décidant à chaque noeud si il faut partir sur le sous-arbre gauche ou le sous-arbre droit. La complexité attendue de l'opération `Search(T, k)` cherchant un élément de clé (*key*) dans un arbre (*tree*) *T* sera donc au maximum proportionnelle à la longueur de la plus longue branche de l'ABR.

## 6.5 Arbres binaire de recherche balancés

L'algorithme d'insertion naïf dans un ABR génère des arbres dont la forme dépend de l'ordre dans lequel les éléments sont insérés. En particulier, si on insère les éléments en ordre croissant,

on fabrique un *peigne* qui n'est ni plus ni moins qu'une liste chaînée. Dans ce cas, on perd tout intérêt à utiliser un ABR.

Il existe deux techniques classiques pour forcer un certain équilibre de l'arbre, sans perdre trop de temps lors de l'insertion : au prix d'un temps constant, on peut re-balancer l'ABR à chaque opération d'insertion. Ces deux techniques sont les *AVL* et les *red-black trees*. Ces deux techniques sont équivalentes du point de vue des performances.

## Performance pour les containers

En résumé, on peut implémenter des containers de bien des façons. En voici quelques unes avec les performances attendues pour chaque opération.

Opération	Tableau non classé	Liste simplement chaînée	Liste doublement chaînée	Arbre binaire de recherche	Table de hachage (version par chaînage)
Search(S,k)	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1 + \alpha)$
Insert(S,k)	$O(1)$	$O(1)$	$O(1)$	$O(\log n)^*$	$O(1)$
Delete(S,k)	$O(1)^*$	$O(n)^*$	$O(1)$	$O(\log n)^*$	$O(1 + \alpha)$
Successor(S,k)	$O(n)$	$O(n)$	$O(n)$	$O(\log n)^*$	$O(n)$
Predecessor(S,k)	$O(n)$	$O(n)$	$O(n)$	$O(\log n)^*$	$O(n)$
Maximum(S,k)	$O(n)$	$O(n)$	$O(n)$	$O(\log n)^*$	$O(n)$
Minimum(S,k)	$O(n)$	$O(n)$	$O(n)$	$O(\log n)^*$	$O(n)$

Opération	Tableau classé	Liste simplement chaînée classée	Liste doublement chaînée classée
Search(S,k)	$O(\log n)$	$O(n)$	$O(n)$
Insert(S,k)	$O(n)$	$O(n)$	$O(n)$
Delete(S,k)	$O(n)$	$O(n)^*$	$O(1)$
Successor(S,k)	$O(1)$	$O(1)$	$O(1)$
Predecessor(S,k)	$O(1)$	$O(n)^*$	$O(1)$
Maximum(S,k)	$O(1)$	$O(1)$	$O(1)$
Minimum(S,k)	$O(1)$	$O(1)^*$	$O(1)$



## Files de priorité et tas

### 8.1 Spécification

On peut donner la spécification ci-dessous pour une file de priorité :

```
Spec PriorityQueue

Operations :
newPriorityQueue : PriorityQueue → PriorityQueue
isEmpty          : PriorityQueue → Bool
length           : PriorityQueue → Int
top               : PriorityQueue → Element
insert            : PriorityQueue Element → PriorityQueue
extractMax       : PriorityQueue → PriorityQueue
maxHeapify        : Array Int → Array
buildMaxHeap     : Array → PriorityQueue

Preconditions :
# To be defined

Axioms :
# To be defined

EndSpec
```

Une file de priorité peut se concevoir en *minimum* ou en *maximum* : les deux propriétés sont parfaitement symétriques. On peut considérer que l'élément de plus haute priorité est celui qui a la clé la plus grande ou la plus petite, au choix. Ici, on a choisi *maximum*, mais si on préfère une file de priorité *minimale*, les opérations `extractMax()` et `maxHeapify` s'appelleront alors `extractMin()` et `minHeapify`.

Les axiomes et les préconditions ne sont pas détaillés.

L'opération `top()` permet d'accéder à l'élément de plus grande priorité. L'opération `insert()` permet d'insérer un élément dans la file de priorité. L'opération `extractMax()` extrait l'élément de plus grande priorité de la file. La spécification ne suppose pas des structures de données mutables, donc cette opération d'extraction retourne la file de priorité modifiée et l'élément extrait

est perdu. En termes de spécification, ce n'est pas un souci puisqu'on dispose de l'opération `top()` qui nous aurait permis d'utiliser cet élément avant de l'extraire.

Enfin l'opération `buildMaxHeap()` permet de convertir un tableau en file de priorité. Selon les choix d'implémentation, ce travail peut être minimal, ou bien relativement conséquent. Quand il est effectué, ce n'est qu'une seule fois pour initialiser la file de priorité à partir d'un ensemble d'éléments.

L'opération `maxHeapify()` est une opération auxiliaire qui est utilisée par `extractMax()` et `buildMaxHeap()` pour restaurer la condition de tas à un certain point de l'arbre lorsqu'elle a pu être détruite.

## 8.2 Implémentation avec une table

On peut réaliser une file de priorité avec une table. Cette table peut être maintenue en ordre complet ou non.

Si on maintient la table en ordre, l'opération d'insertion d'un élément à sa place dans la table ordonnée prendra un temps linéaire  $O(n)$ . L'opération d'accès à l'élément de priorité maximale pourra être traitée par dichotomie pour un accès en  $O(\log n)$ .

Si on ne maintient pas la table en ordre, l'insertion peut se faire à la fin de la table en temps constant  $O(1)$ . L'accès à l'élément de priorité maximale nécessitera une recherche en temps linéaire  $O(n)$ .

L'extraction de l'élément de priorité maximale se fera en temps linéaire  $O(n)$  dans les deux cas, puisqu'il faudra supprimer l'élément de la table.

Tout algorithme exploitant une file de priorité réalisera des opérations d'insertion et d'extraction. Il s'agit donc de trouver le meilleur compromis en fonction du nombre de chacune de ces opérations.

## 8.3 Tas

L'implémentation d'une file de priorité à l'aide d'une table est rarement la solution la plus efficace. On peut faire beaucoup mieux en remarquant qu'on peut maintenir un certain ordre entre les éléments de la file de priorité qui nous garantisse un accès très rapide – en  $O(1)$  – à l'élément de plus haute priorité, sans pour autant nécessiter de classer complètement tous les éléments.

Un tas est une structure de données qui maintient un tel ordre partiel entre ses éléments. Cet ordre partiel est suffisant pour réaliser efficacement les opérations dont nous avons besoin. En ordonnant partiellement les éléments, on économise une partie du calcul par rapport à un ordonnancement complet. On économise également par rapport à aucun ordonnancement (cf. les deux cas précédents avec la table). On réalise donc un *compromis* intéressant.

Un tas maximal (resp. minimal) est un arbre binaire presque complet qui respecte les conditions suivantes :

- l'élément à la racine est plus grand (resp. petit) que son fils gauche et son fils droit
- les sous-arbres gauche et droit sont eux-mêmes des tas maximaux (resp. minimaux).

De façon intuitive, un arbre binaire presque complet est un arbre binaire parfait auquel il manque quelques feuilles :

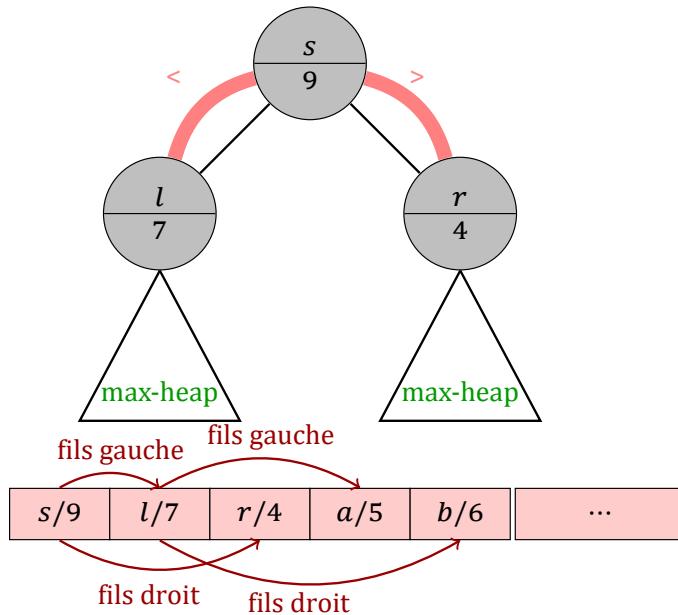
La particularité d'un tas maximal (resp. minimal) est qu'il permet d'accéder en  $O(1)$  au plus grand (resp. plus petit) élément, puisque celui-ci est à la racine de l'arbre. Il permet également

de réaliser les opérations d'insertion et d'extraction en temps  $O(\log n)$  puisque celles-ci vont se dérouler le long d'une branche de l'arbre.

Attention : ne pas confondre avec les propriétés d'un arbre binaire de recherche. Ce sont des conditions très différentes.

## 8.4 Implémentation à l'aide d'un tableau

La particularité pour un tas d'être un arbre binaire *presque complet* permet l'implémentation à l'aide d'un tableau selon le schéma de codage suivant :



Les opérations d'accès sont définies par :

```

function Left(i)
    return 2*i+1
end

function Right(i)
    return 2*i+2
end

function Parent(i)
    return i/2
end

```

Effectivement, le tableau sera rempli *sans trous* parce que l'arbre binaire est presque complet. Les opérations d'accès sont parfaitement définies et sont en  $O(1)$ .

Remarque : on sait même où sont situées les feuilles de l'arbre. Tous les noeuds situés à un indice inférieur à la moitié de la longueur du tableau ont des fils. Les autres n'en ont pas.

## 8.5 Construction d'un tas à partir d'un tableau

Nous allons définir l'opération auxiliaire `maxHeapify()`. Cette opération suppose que l'on dispose d'un arbre binaire presque complet dans un tableau. Elle suppose également que les sous-arbres gauche et droit sont des tas. Le rôle de cette fonction est de rétablir la condition de tas à la racine si cette condition n'est pas respectée.

```

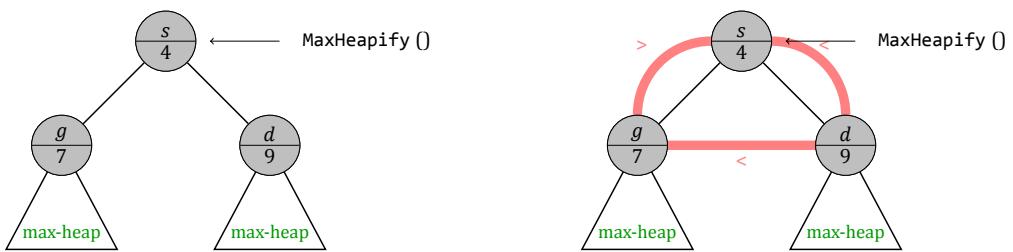
function Swap(A, i, j)
    tmp = A[i]
    A[i] = A[j]
    A[j] = tmp
end

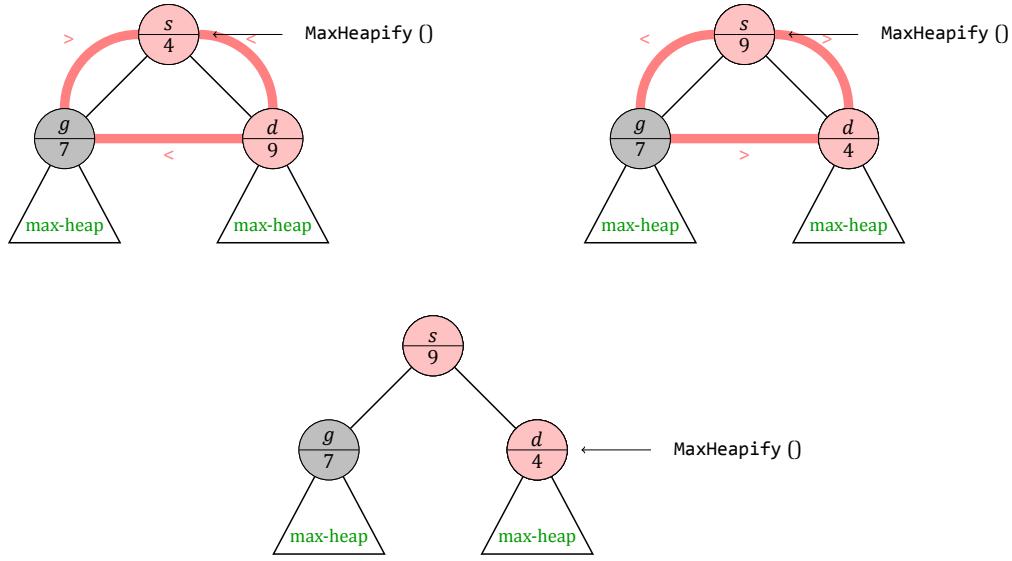
function MaxHeapify(A, i)
    l <- Left(i)
    r <- Right(i)
    if l < heapSize && A[l] > A[i]
        largest <- l
    else
        largest <- i
    if r < heapSize && A[r] > largest
        largest <- r

    if largest != i
        Swap(A, i, largest)
        MaxHeapify(A, largest)
    end
end

```

Cette fonction compare la valeur du noeud à la racine avec la valeur de chacun de ses noeuds fils. Elle sélectionne la plus grande valeur (plus petite si c'est un tas minimal) des trois valeurs. Si ce n'est pas celle à la racine, elle la permute avec la valeur à la racine. Si une permutation de cette nature a lieu, potentiellement, la condition de tas maximal est détruite pour le sous-arbre qui a subi la permutation. Il faut donc appeler récursivement la fonction `maxHeapify()` sur ce sous-arbre pour restaurer cette condition.





Avec cette opération `maxHeapify()` il devient facile de construire un tas maximal à partir d'un tableau non-ordonné. En utilisant la remarque selon laquelle les feuilles sont dans la seconde moitié du tableau, il suffit de balayer le tableau de droite à gauche. Chaque feuille représente tri-vialement un tas de hauteur 0. En appliquant `maxHeapify()` à partir du parent commun à deux feuilles, on fabrique un nouveau tas de hauteur 1. Après avoir construit des tas de hauteur 1, on peut construire ceux de hauteur 2 et ainsi de suite.

Cet algorithme s'exprime de la façon suivante :

```
function BuildMaxHeap(A)
    heapSize = A.length
    for i <- (heapSize-1)/2 downto 0
        MaxHeapify(a, i)
    end
```

## 8.6 Complexité des opérations



## Union-find

On suppose disposer d'une ensemble de groupes d'éléments. La structure Union-Find vise à réaliser deux opérations particulières dans le meilleur temps possible :

1. trouver à quel groupe appartient un élément,
2. réunir deux groupes en un seul.

Il est facile de réaliser une implémentation naïve de cette structure :

```
# C est un dictionnaire de classes de noeuds.
# Tous les noeuds ont un représentant, initialement eux-mêmes.

# Trouver un représentant de la classe
def naive_find(C, u):
    while C[u] != u:
        u = C[u]
    return u

def naive_union(C, u, v):
    # Trouver les deux représentants
    u = naive_find(C, u)
    v = naive_find(C, v)
    # Les réunir : l'un réfère à l'autre
    C[u] = v

C = { 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8}
naive_union(C, 2, 7)
naive_union(C, 1, 5)
naive_union(C, 7, 1)
print(C)
```

Le problème de cette approche naïve réside dans la manière de traiter les représentants de chaque classe. On crée des chaînes de références qui peuvent aller en s'allongeant d'une manière non-optimale.

Ce problème peut-être résolu par une technique dite de *compression des chemins* qui consiste essentiellement à mettre à jour le représentant de chaque groupe.

```
# C est un dictionnaire de classes de noeuds.
# Tous les noeuds ont un représentant, initialement eux-mêmes.
def find(C, u):
    if C[u] != u:
        # Compression du chemin
        C[u] = find(C, C[u])
    return C[u]

# Ici C est un dictionnaire des classes des noeuds
# et R indique le rang d'un noeud
def union(C, R, u, v):
    u, v = find(C, u), find(C, v)
    # Union par le rang du noeud
    if R[u] > R[v]:
        C[v] = u
    else:
        C[u] = v
    # Même rang, il faut monter v d'un niveau
    if R[u] == R[v]:
        R[v] += 1

C = { 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8}
R = { u: 0 for u in C }
naive_union(C, 2, 7)
naive_union(C, 1, 5)
naive_union(C, 7, 1)
```

# Graphes

## 10.1 Définitions

Un graphe est défini par la donnée d'un couple  $(V, E)$ , où  $V$  est un ensemble de *sommets* et  $E \subseteq V \times V$  est un ensemble d'*arêtes*. Les dénominations  $V$  et  $E$  viennent de l'anglais :  $V$  pour *vertices* et  $E$  pour *edges*.

Deux noeuds  $u$  et  $v$  sont *adjacents* si et seulement si  $(u, v) \in E$  ou  $(v, u) \in E$ .

On peut donner des noms à différentes types de graphes selon leurs propriétés :

Si  $\forall (u, v) \in E \quad (v, u) \in E$  on dit que le graphe est *non-orienté*, sinon il est *orienté*.

Soit  $w : E \rightarrow \mathbb{R}$  une fonction de *poids*.  $G = (V, E, w)$  définit un graphe *pondéré*.

On définit le *degré entrant* (resp. *sortant*) d'un noeud :  $\deg_{\text{in}}(u) = |\{(v, u) \in E\}|$  et  $\deg_{\text{out}}(u) = |\{(u, v) \in E\}|$ .

Un *chemin* de  $u$  à  $v$  est une séquence de noeuds  $u_0, u_1, \dots, u_n$  tels que  $u = u_0, v = u_n$  et  $\forall i \in [1, n] \quad (u_{i-1}, u_i) \in E$ . Le noeud  $v$  est accessible depuis  $u$  si il existe un chemin de  $u$  à  $v$ .

La *longueur* d'un chemin est la somme des poids des arêtes composant ce chemin.

Un *cycle* est un chemin reliant un noeud à lui-même.

Un graphe est *complet* si  $E = V \times V$ .

Un graphe non-orienté est *connexe* si tous les noeuds sont reliés entre eux (il existe toujours un chemin).

Un graphe orienté est *fortement connexe* si tous les noeuds sont reliés entre eux :  $\forall (u, v) \in V \times V \quad v$  accessible depuis  $u$ .

Un *arbre* est un graphe non-orienté, connexe et acyclique. Un arbre peut être doté d'une *racine*  $r \in V$ .

Une *forêt* est un graphe orienté, acyclique (DAG pour *Directed Acyclic Graph*) dans lequel chaque noeud possède au plus un unique antécédent.

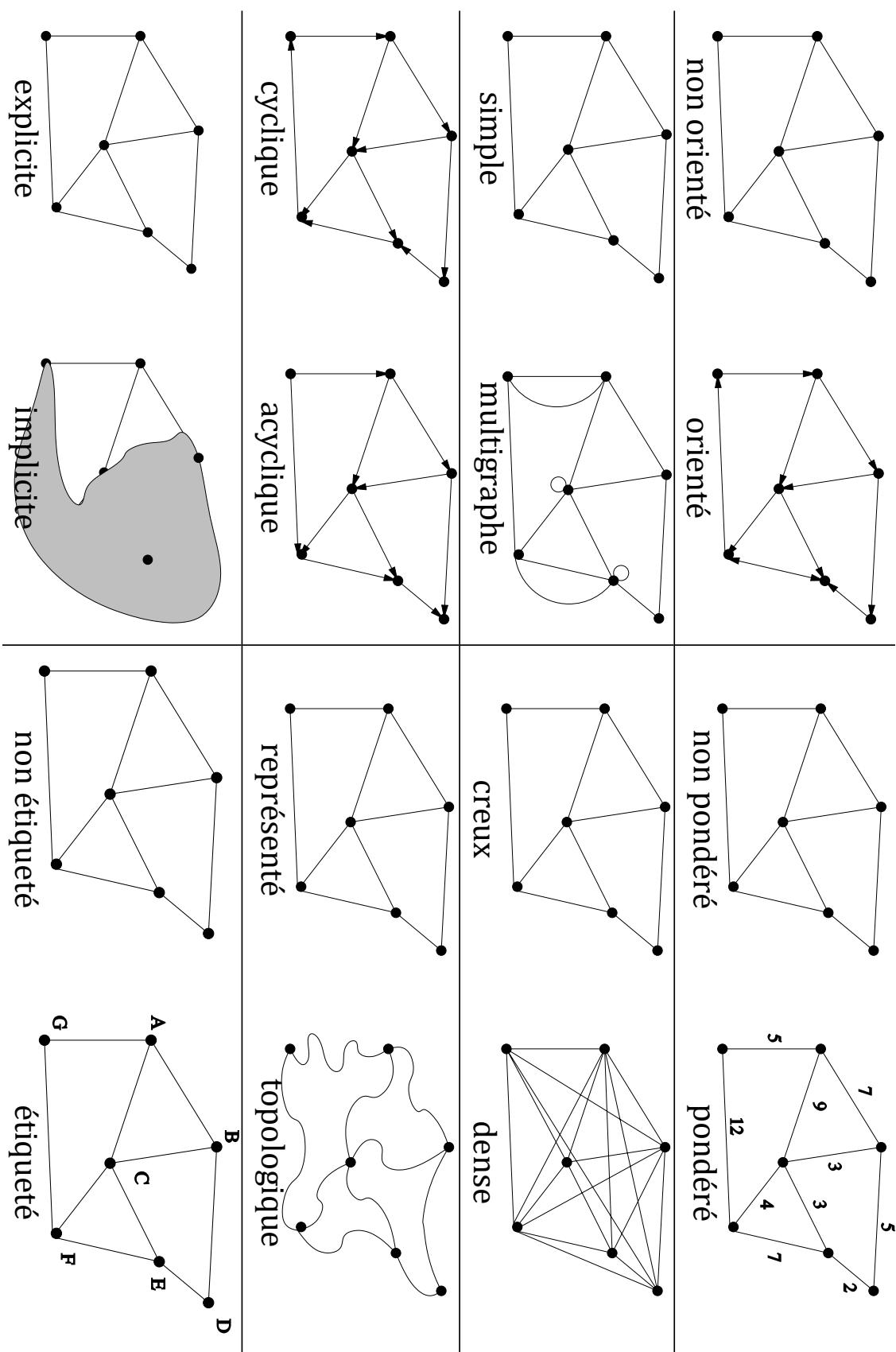
Une *arborescence* est une forêt à une seule racine.

On appelle *racine* un noeud n'ayant pas d'antécédent.

On appelle *feuille* un noeud n'ayant pas de successeur.

On appelle *profondeur* d'un noeud la longueur du chemin depuis la racine.

On appelle *hauteur* de l'arbre la profondeur maximale d'un noeud dans l'arbre.



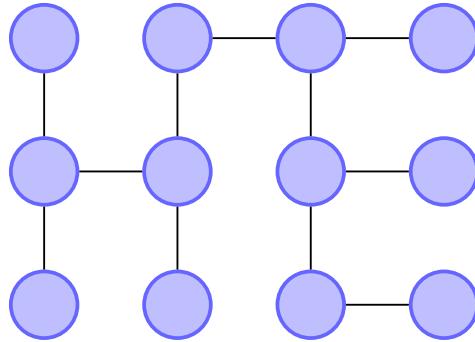


FIGURE 10.1 – exemple d’arbre

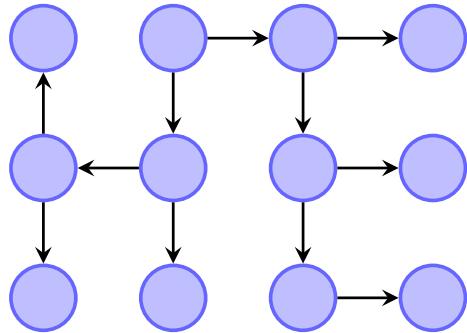


FIGURE 10.2 – exemple d’arborescence

Tout noeud est accessible depuis la racine par un unique chemin.

- L’accessibilité se prouve par l’absurde : si un noeud n’a pas de parent, c’est la racine ; s’il en a un, le parent doit ne pas être accessible : on itère (et le graphe est fini).
- L’unicité se prouve simplement par récursion (le chemin vers le parent est unique)

Arbres particuliers :

- *Arbre binaire* : chaque noeud a au plus 2 successeurs. Leur hauteur  $h$  vérifie  $h \leq |N| \leq 2^h - 1$
- *Arbre équilibré* : les sous-arbres sont balancés et la différence de hauteur entre les sous-arbres n’excède pas 1
- *Arbre complet* : toutes les feuilles ont pour profondeur  $h$
- *Arbre presque complet* : toutes les feuilles  $f_1, \dots, f_p$  sont à la profondeur  $h$  jusqu’au rang  $1 \leq k \leq p$ , et à la profondeur  $h - 1$  à partir du rang  $k + 1$ .

## 10.2 Implémentation par matrice d’adjacence

Une solution simple et efficace pour représenter un graphe consiste à numérotter ses  $n$  noeuds ( $0 \dots n - 1$ ) et à représenter l’ensemble des arêtes à l’aide d’une matrice de taille  $n \times n$ .

Lorsqu’on représente un graphe non-orienté, la matrice d’adjacence est symétrique.

Lorsqu’on représente un graphe pondéré, on peut utiliser la matrice d’adjacence pour stocker les poids des arêtes. Il faut toutefois faire attention : il faut une valeur conventionnelle pour indiquer qu’il n’y a pas d’arête entre deux noeuds. Si tous les poids sont positifs, on peut par

exemple choisir une valeur de  $-1$  pour signifier que deux noeuds ne sont pas adjacents. En revanche, si toutes les valeurs sont permises pour les poids, il faudra une matrice d'adjacence et une matrice de poids.

L'espace occupé par cette représentation est  $\Theta(|V|^2)$ .

### 10.3 Implémentation par liste d'adjacence

Une autre solution pour représenter un graphe consiste à utiliser des listes d'adjacence : pour chaque noeud du graphe, on indique les noeuds qui lui sont adjacents dans une liste.

Les éléments de la liste d'adjacence de  $u$  peuvent être des paires  $(v, d)$  où  $d$  est la distance de  $u$  à  $v$  dans le cas d'un graphe pondéré.

L'espace occupé par cette représentation est composé d'une part par la liste des noeuds du graphe, d'autre part par la somme des longueurs des listes d'adjacence : il est donc en  $\Theta(|V| + |E|)$ .

Selon la densité (nombre d'arêtes du graphe par rapport au nombre maximal d'arêtes soit  $|V|^2$ ), cette représentation est plus ou moins économique que la représentation par matrice d'adjacence. Pour un graphe complet, elle est moins économique. Pour un graphe peu dense, elle est plus économique.

Pour le choix de la représentation, il faut aussi tenir compte de l'algorithme que l'on implémente : selon sa structure, une représentation par matrice d'adjacence ou par liste d'adjacence sera préférable. Si la structure de l'algorithme est du type :

```
for u in V
    for v in Adj(u)
        ...
```

une représentation par liste d'adjacence sera recommandée, car elle suivra la structure de l'algorithme.

En revanche, nous verrons que pour l'algorithme de Floyd-Warshall, la représentation par matrice d'adjacence est nécessaire.

### 10.4 Implémentation par liste d'incidence

Une dernière solution pour représenter un graphe consiste à simplement utiliser une liste d'incidence qui n'est autre que la liste des arêtes. Cette représentation va bien entendu utiliser un espace en  $\Theta(|E|)$ . Cette représentation n'est adaptée qu'à certains algorithmes qui n'ont besoin que de parcourir la liste des arêtes sans accès particulier par un sommet du graphe.

## En Python

### 11.1 Piles et files

Python fournit une collection standard nommée `deque` (prononcer *deck*) qui est une généralisation des piles et des files. C'est en fait une structure linéaire à double entrée permettant d'ajouter des éléments à un bout ou à l'autre.

Les opérations `enqueue`, `dequeue`, `push` et `pop` sont garanties en  $O(1)$  (modulo l'extensibilité de la structure).

Se référer à <https://docs.python.org/3.5/library/collections.html#deque-objects> pour plus de précisions.

Sur cette base, on peut très simplement implémenter :

```
from collections import deque

class Stack:
    def __init__(self):
        self._items = deque()
    def push(self, item):
        self._items.append(item)
    def pop(self):
        return self._items.pop()
    def __nonzero__(self):
        return bool(self._items)
```

On remarquera la légère différence par rapport à notre spécification : ici, l'implémentation de `pop` travaille par effet de bord : elle retourne la valeur du sommet de pile, tout en modifiant le contenu de la pile elle-même. On se passe ainsi de l'opération `top`.

L'erreur de *pile vide* est gérée par la structure `deque` sous-jacente. Par défaut, les `deque` sont extensibles, on n'a donc pas à s'inquiéter d'un éventuel débordement de capacité.

Exemple d'utilisation :

```
f = Stack()
f.push(3)
f.push(2)
f.push(1)
```

```
while not f:
    print f.pop(),
# >>> 3 2 1
```

De la même façon, on peut implémenter une file :

```
from collections import deque

class Queue:
    def __init__(self):
        self._items = deque()
    def enqueue(self, item):
        self._items.appendleft(item)
    def dequeue(self):
        return self._items.pop()
    def __nonzero__(self):
        return bool(self._items)
```

Example d'utilisation :

```
f = Queue()
f.enqueue(3)
f.enqueue(2)
f.enqueue(1)
while not f:
    print f.dequeue(),
# >>> 3 2 1
```

## 11.2 Listes et tuples Python

Les listes Python *ne sont pas* des listes chaînées. Les listes Python sont des tables extensibles. La différence entre table et liste chaînée est visible sur l'opération `index[ ]` : elle est en  $O(1)$  pour une table et en  $O(n)$  pour une liste chaînée. Le comportement des listes Python est résumé dans le tableau ci-dessous qui montre des différences majeures avec celui des listes chaînées :

Opération	Exemple	Complexité	Notes
Index	<code>l[i]</code>	$O(1)$	
Store	<code>l[i] = 0</code>	$O(1)$	
Length	<code>len(l)</code>	$O(1)$	
Append	<code>l.append(5)</code>	$O(1)$	
Pop	<code>l.pop()</code>	$O(1)$	
Clear	<code>l.clear()</code>	$O(1)$	identique à <code>l = []</code>
Slice	<code>l[a:b]</code>	$O(b - a)$	<code>l[1:5]</code> est en $O(1)$ , <code>l[:]</code> est en $O(\text{len}(l) - 0) = O(n)$
Extend Construction	<code>l.extend(...)</code> <code>list(...)</code>	$O(\text{len}(...))$ $O(\text{len}(...))$	dépend de la longueur de l'extension dépend de la longueur de l'argument
check =, !=	<code>l1 == l2</code>	$O(n)$	
Insert	<code>l[a:b] = ...</code>	$O(n)$	
Delete	<code>del l[i]</code>	$O(n)$	
Remove	<code>l.remove(...)</code>	$O(n)$	
Containment	<code>x in/not in l</code>	$O(n)$	recherche dans la liste
Copy	<code>l.copy()</code>	$O(n)$	identique à <code>l[:]</code> qui est en $O(n)$
Pop	<code>l.pop(0)</code>	$O(n)$	
Extreme value	<code>min(l)/max(l)</code>	$O(n)$	
Reverse	<code>l.reverse()</code>	$O(n)$	
Iteration	<code>for v in l :</code>	$O(n)$	
Sort	<code>l.sort()</code>	$O(n \log n)$	indépendant de key/reverse
Multiply	<code>k*l</code>	$O(kn)$	$5 * l$ est en $O(n)$ , <code>len(l)*l</code> est en $O(n^2)$

Les tuples supportent toutes les opérations des listes qui ne mutent pas la structure de donnée et ce avec la même complexité.

### 11.3 Dictionnaires

Les dictionnaires font partie des structures offertes nativement par Python. La complexité des opérations fournies est résumée ci-dessous.

Opération	Exemple	Complexité	Notes
Index	<code>d[k]</code>	$O(1)$	
Store	<code>d[k] = v</code>	$O(1)$	
Length	<code>len(d)</code>	$O(1)$	
Delete	<code>del d[k]</code>	$O(1)$	
get/setdefault	<code>d.method</code>	$O(1)$	
Pop	<code>d.pop(k)</code>	$O(1)$	
Pop item	<code>d.popitem()</code>	$O(1)$	
Clear	<code>d.clear()</code>	$O(1)$	identique à <code>s = {}</code> ou <code>s = dict()</code>
Views	<code>d.keys()</code>	$O(1)$	
Construction	<code>dict(...)</code>	<code>len(...)</code>	
Iteration	<code>for k in d :</code>	$O(n)$	toutes les formes : clés, valeurs, items

Le principe de construction des dictionnaires est intéressant et une réalisation en est donnée en annexe. Toutefois, l'approche simpliste proposée doit être améliorée pour obtenir de bonnes performances en pratique.

## 11.4 Files de priorité

Python fournit le module `heapq` qui implémente une file de priorité *minimale*. Cette file de priorité est réalisée par une structure de tas. Les complexités des opérations sont conformes aux attentes : `top()` en  $O(1)$ , `heappush()` et `heappop()` en  $O(\log n)$  et `heapify()` en  $O(n)$ .

<https://docs.python.org/3.5/library/heappq.html>

On peut réaliser soi-même ce module en Python. Un exemple est donné en annexe.

## 11.5 Ensembles

Python fournit la structure de donnée `Set`, pratique et plus spécifique que nos *containers* car elle supporte les opérations ensemblistes.

Attention : cette structure ne correspond pas à la structure Union-Find nécessaire à l'implémentation de l'algorithme de Kruskal. L'algorithme de Kruskal nécessite l'opération `FindSet()` qui retrouve l'ensemble auquel un élément appartient.

Opération	Exemple	Classe	Notes
Length	<code>len(s)</code>	$O(1)$	
Add	<code>s.add(5)</code>	$O(1)$	
Containment	<code>x in/not in s</code>	$O(1)$	à comparer avec list/tuple en $O(n)$
Remove	<code>s.remove(5)</code>	$O(1)$	à comparer avec list/tuple en $O(n)$
Discard	<code>s.discard(5)</code>	$O(1)$	
Pop	<code>s.pop()</code>	$O(1)$	à comparer avec list en $O(n)$
Clear	<code>s.clear()</code>	$O(1)$	identique à <code>s = set()</code>
Construction	<code>set(...)</code>	<code>len(...)</code>	
check =, !	<code>s!=t</code>	$O(\min(\text{len}(s), \text{len}(t)))$	
<code>&lt;=/&lt;</code>	<code>s &lt;= t</code>	$O(\text{len}(s))$	<code>issubset</code>
<code>&gt;=/&gt;</code>	<code>s &gt;= t</code>	$O(\text{len}(s))$	<code>issuperset</code>
Union	<code>s</code>	$t$	$O(\text{len}(s) + \text{len}(t))$
Intersection	<code>s &amp; t</code>	$O(\min(\text{len}(s), \text{len}(t)))$	
Difference	<code>s - t</code>	$O(\text{len}(t))$	
Symmetric	<code>s ^ t</code>	$O(\text{len}(s))$	
Diff			
Iteration	<code>for v in s :</code>	$O(n)$	
Copy	<code>s.copy()</code>	$O(n)$	

Les opérations sont plus rapides qu'avec les listes car il n'est pas nécessaire de maintenir les objets dans un ordre donné (à vérifier).

## 11.6 Arbres binaires de recherche

Les arbres binaires de recherche ne sont pas fournis par Python. On trouve facilement des implémentations correspondant à notre description. En voici une parfaitement illustrée : <http://www.laurentluce.com/posts/binary-search-tree-library-in-python/>

Bien sûr, pour des performances acceptables, il faut affiner cette implémentation et balancer l'arbre lors des opérations d'insertion et de suppression. Ceci peut se faire soit par la technique des arbres AVL, soit par la technique des arbres colorés rouge-noir (*red-black trees*).

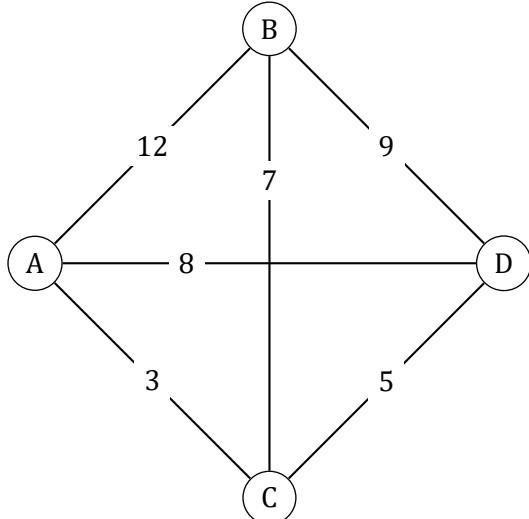
## 11.7 Graphes

Python comme la plupart des langages n'offre pas de support natif pour représenter les graphes. Il faut donc s'en remettre aux représentations suggérées à la section Graphes.

Il faut distinguer le cas selon que l'on veut accéder aux noeuds par leur nom ou étiquette, ou bien que l'on numérote les noeuds pour y accéder ensuite par leur numéro. Dans le premier cas, il faut se reposer sur des dictionnaires Python. Dans le second cas, on peut se reposer sur des listes Python (tableaux) puisque l'index est un nombre dans l'intervalle  $[0..|V| - 1]$ .

Les matrices d'adjacences sont facilement implémentées par des listes de listes ou des dictionnaires de dictionnaires.

Exemple d'un graphe orienté et pondéré à 4 noeuds :



Implémentation par listes d'adjacence avec accès par le nom du noeud :

```

G = {'a' : {'b' : 12, 'd' : 8},
      'b' : {'c' : 7 },
      'c' : {'a' : 3, 'd' : 5 },
      'd' : {'b' : 9 } }
  
```

Le même graphe représenté par matrice d'adjacence avec accès par le numéro du noeud :

```

G = [[None, 12 , None, 8 ],
      [None, None, 7 , None],
      [3 , None, None, 5 ],
      [None, 9, None, None]]
```

---

Il est à noter que Python offre une facilité pour les graphes pondérés. La valeur `None` peut être utilisée en parallèle de toute valeur numérique pour indiquer que deux noeuds ne sont pas adjacents. Ceci est possible parce que Python ne restreint pas le type des objets qui sont stockés dans une liste ou un dictionnaires. Peu d'autres langages tolèrent ceci.

Des bibliothèques ont été développées pour aider à la gestion efficace et à la visualisation de grands graphes.

**Troisième partie**

**Techniques fondamentales**



## Problèmes

### 12.1 Définition

Un **problème** est défini par

- un domaine de définition pour des données en entrée
- un domaine de définition pour des données en sortie
- une relation entre données en entrée et données en sortie

Une **instance** d'un problème est définie par une valeur particulière des données en entrée.

Une **solution** à une instance d'un problème est définie par une valeur des données de sortie telle que la relation entre la valeur des données en entrée pour cette instance et cette valeur des données de sortie soit satisfaite.

Résoudre un problème ou trouver une **solution à un problème** consiste à définir un algorithme qui pour toute entrée, calcule une solution au problème quand elle existe.

La résolution d'un problème est différente de son **optimisation** qui est plus difficile !

On parle de **qualité** d'une solution : certaines solutions peuvent être moins couteuses que d'autres

Lorsqu'on cherche à résoudre un problème, on peut s'intéresser à :

- trouver une solution arbitraire, éventuellement rapidement
- trouver une solution optimale.

On peut dégager des techniques génériques sur lesquelles construire un algorithme pour résoudre un problème donné.



## Méthode Glouton

### 13.1 Définition

Une méthode est qualifiée de glouton lorsqu'elle construit la solution au problème incrémentalement et de façon monotone, sans jamais considérer de choix alternatifs et sans jamais revenir en arrière.

Une méthode glouton ne garantit rien en terme d'optimalité : le choix à chaque étape se basant sur des informations locales ne permet d'atteindre en général qu'un optimum local.

### 13.2 Exemple du classement par insertion

On se reportera à la définition du problème du classement et à l'algorithme de classement par insertion.

C'est un algorithme glouton par essence. Ici, il n'existe qu'une solution au problème, il n'y a donc pas de problème d'optimalité.

La technique de classement par tas peut également être qualifiée de glouton : elle sort les éléments du tas un par un, en remettant le tas en ordre à chaque fois. C'est une technique glouton plus sophistiquée que celle du classement par insertion.

### 13.3 Exemple du rendu de monnaie

#### Illustration

Dans un système monétaire, on a défini des pièces qui ont chacune une valeur faciale. Le problème du rendu de monnaie consiste à déterminer la décomposition d'un montant donné en un nombre minimal de pièces.

Instance : rendre 2€72 avec des pièces de 1 et 2 euros et de 1, 2, 5, 10, 20 et 50 centimes.

Solutions :

$2\text{€} + 50\text{¢} + 20\text{¢} + 2\text{¢}$  ✓

$1\text{€} + 1\text{€} + 20\text{¢} + 20\text{¢} + 20\text{¢} + 10\text{¢} + 1\text{¢} + 1\text{¢}$  ✓

$1\text{€} + 50\text{¢} + 4 \times 20\text{¢} + 3 \times 10\text{¢} + 1\text{¢}$  X

### Définition formelle du problème

Données en entrée :

Un système monétaire défini par  $n \in \mathbb{N}$  le nombre de valeurs faciales et  $\mathcal{S} \in \mathbb{R}^n$  l'ensemble des valeurs faciales, ainsi que  $s \in \mathbb{R}$  la somme à rendre.

Données en sortie :

Un n-uplet  $t \in \mathbb{N}^n$  indiquant le nombre de chaque pièces.

Relation entre les données d'entrée et les données de sortie :

$$s = \sum_{i=1}^n t_i \times S_i$$

où  $t_i$  désigne le nombre de pièces de valeur faciale  $S_i$ .

Solution :  $t \in \mathbb{N}^n$  est une solution au problème de rendu de monnaie défini par  $(n, \mathcal{S}, s)$  si et seulement si la relation

$$s = \sum_{i=1}^n t_i \times S_i$$

est vérifiée.

Dans notre cas :

$n = 8, \mathcal{S} = (2, 1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01)$  et  $s = 2.72$ .

### Qualité d'une solution

On considère le nombre de pièces utilisées pour réaliser la somme choisie. Coût : on note  $\text{cout}(t) = \sum_{i=1}^n t_i$  le coût de la solution  $t$ .

### Algorithme glouton

On suppose  $\mathcal{S}$  trié par ordre décroissant. Pour chacune des valeurs faciales prises en ordre décroissant, on retire autant de fois que possible cette valeur faciale de la somme à réaliser.

```

function monnaie(S, s)
    t ← [0, ... 0]
    i ← 1
    r ← s
    while r > 0 and i ≤ n
        if r - S[i] ≥ 0
            t[i] ← t[i] + 1
            s ← s - S[i]
        else
            i ← i + 1
        return t
    end

```

Cet algorithme est incomplet : il peut ne pas trouver de solution, même lorsqu'il en existe une. Exemple :  $S = [3, 2]$  et  $s = 4$ .

Cet algorithme ne trouve pas toujours la solution optimale. Exemple :  $S = [4, 3, 1]$  et  $s = 6$ . L'algorithme glouton donne  $s = 4 + 1 + 1$  alors qu'on peut trouver  $s = 3 + 3$ .

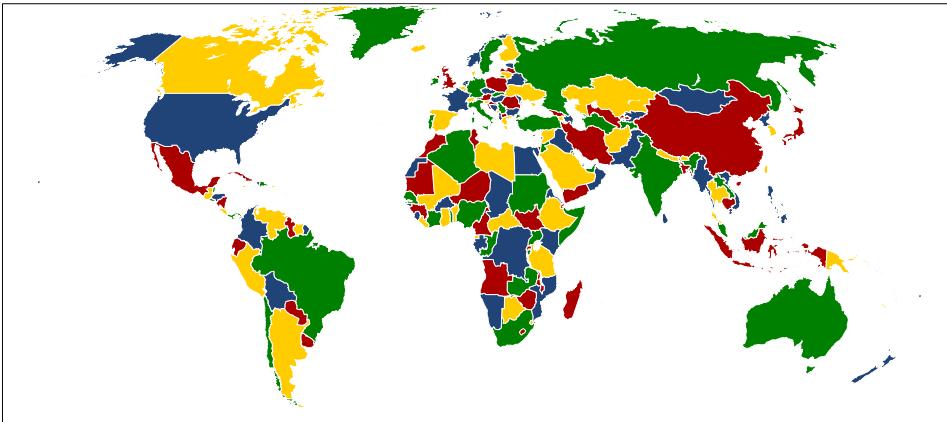
On qualifie de *canonique* un système pour lequel l'algorithme glouton donne la solution optimale. La plupart des systèmes monétaires sont aujourd'hui canoniques. Celui du Royaume Uni ne l'était pas avant la **décimalisation** (1971).

Il existe un algorithme en  $O(n^3)$  pour tester si un système monétaire à  $n$  valeurs faciales est canonique.

## 13.4 Exemple de la coloration de graphe

### Illustration

On a cherché dans le passé à minimiser le nombre de couleurs à utiliser pour colorer les différentes régions de cartes géographiques. La contrainte dans ce cas est que deux régions adjacentes ne peuvent recevoir la même couleur, sans quoi on ne peut les distinguer. Il est apparu assez rapidement qu'il semblait suffire de 4 couleurs pour colorer n'importe quelle carte où les régions sont connexes.



### Définition du problème

On se reportera aux définitions relatives aux graphes plus haut.

Données en entrée : un graphe  $G = (V, E)$ , un ensemble  $C$  de  $n_c$  couleurs.

Données en sortie : une fonction  $\text{col} : V \rightarrow C$  qui associe une couleur à chaque noeud.

Relation entre données d'entrée et de sortie :

$$\forall (v_i, v_j) \in V \times V \quad v_i \text{ adjacent à } v_j \Rightarrow \text{col}(v_i) \neq \text{col}(v_j).$$

Une fonction  $\text{col} : V \rightarrow C$  est une solution au problème si et seulement si la relation ci-dessus est vérifiée.

L'espace des solutions possibles a pour taille  $C^{|V|}$ , il est donc exponentiel par rapport au nombre de noeuds du graphe. Tout algorithme procédant par énumération des solutions possibles sera rapidement limité.

### Qualité d'une solution

Ici, on définit la qualité d'une solution par le nombre de couleurs utilisées pour colorer le graphe.

Ici, la qualité du solution est définie par le nombre de couleurs nécessaires à colorer le graphe : plus ce nombre est faible, meilleure est la solution.

### Algorithme glouton

On peut utiliser une méthode glouton très simple pour colorer un graphe.

Supposons qu'on ait au départ autant de couleurs que de noeuds dans le graphe, de façon à être certain qu'une solution existe. Bien sûr si nous n'avons pas assez de couleurs, le problème peut ne pas avoir de solution.

Considérons l'ensemble des couleurs effectivement utilisées, que nous réduisons à la seule première couleur disponible. Parcourons chaque noeud du graphe :

- si ce noeud peut être coloré sans conflit avec une des couleurs déjà utilisée, alors choisissons cette couleur,
- sinon, augmentons l'ensemble des couleurs utilisées par une nouvelle couleur, et colorons le noeud courant avec cette couleur.

Cet algorithme est correct : il associe toujours une couleur sans conflit à un noeud et ne crée pas de conflit en associant une nouvelle couleur. Il y a autant de couleurs que de noeuds, donc nous sommes certains de trouver une solution.

Formellement, l'algorithme peut s'écrire :

```

function checkConstraints(g, u, c):
    #c Vérification que les contraintes
    #c sont respectées pour u
    for v in g[u]
        if v in c & c[u] = c[v]
            return False
    end
    return True
end

function greedyColor(g)
    c ← 0
    #c Initialisation du dictionnaire
    #c de couleurs
    color ← {}
    for vi in sorted(g)
        k ← 0
        color[vi] ← k
        while k < c & checkConstraints(g, vi, color) = False
            k ← k+1
            color[vi] ← k
            print(c, k, color)
        end
        if k = c
            #c Besoin d'une couleur supplémentaire
            color[vi] ← c
    end

```

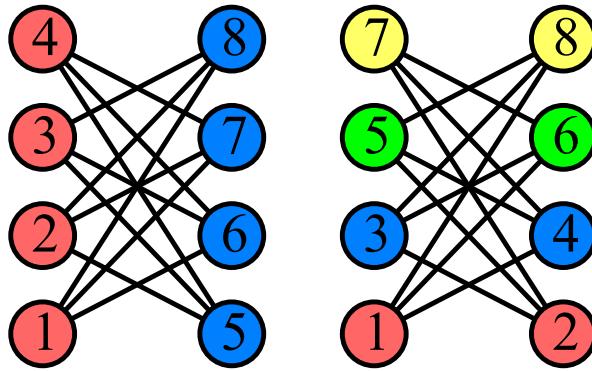
```

    c ← c+1
end
end
return c
end

```

Ici, nous n'avons pas mis de borne au nombre de couleurs que l'algorithme pourra utiliser. Nous supposons que de nouvelles couleurs sont disponibles à volonté. Nous pouvons remarquer que nous n'avons pas besoin de  $|V|$  couleurs, mais seulement de  $|\Delta(G)| + 1$  couleurs où  $|\Delta(G)|$  est le degré du graphe, c'est-à-dire le plus grand nombre de voisins qu'un noeud peut avoir. Ce nombre est effectivement suffisant : il suffit de parcourir les noeuds du graphe par degré croissant. Nous disposerons toujours de couleurs en nombre suffisant, même si nous devons ajouter une nouvelle couleur à chaque itération.

Cet algorithme ne fournit pas toujours une solution optimale comme l'exemple suivant le montre :



Selon l'ordre de parcours des noeuds du graphe, l'algorithme glouton va renvoyer des solutions de qualité très différentes. Dans cet exemple, les numéros indiquent l'ordre de parcours des noeuds.

## 13.5 Glouton et optimalité

On peut voir sur les deux exemples précédents que l'algorithme glouton ne fournit pas toujours une solution optimale.

Tous les systèmes monétaires ne sont pas canoniques et l'algorithme glouton ne donne pas toujours la solution optimale sur le problème du rendu de monnaie. Il existe un [algorithme](#) pour tester en  $o(n^3)$  si un système monétaire est canonique.

Pour le problème de la coloration de graphe, il n'y a pas d'algorithme glouton qui garantisse l'optimalité de la solution.

Il existe une structure de problème qui permet de garantir qu'un algorithme glouton donnera la solution optimale. Une esquisse de l'analyse de cette structure est proposée sur ce [site](#) pour les curieux.



## Méthode diviser pour régner

### 14.1 Définition

La technique *diviser pour régner* repose sur un principe de récurrence :

- on ne sait pas résoudre facilement directement le problème
- on sait résoudre les petites instances du problème
- on sait combiner les solutions à plusieurs petites instances du problème pour fournir une solution à une instance plus grande.

Le principe général de cette technique peut donc se résumer par :

- si la taille est inférieure à la taille minimale  $n_0$ , alors donner la solution
- si la taille est supérieure à la taille minimale  $n_0$ , alors
  - diviser le problème en 2 sous-problèmes de taille  $\frac{n}{2}$
  - résoudre récursivement les 2 sous-problèmes
  - combiner les 2 solutions aux sous-problèmes en une solution au problème original.

Les valeurs de 2 sous-problèmes de taille  $\frac{n}{2}$  ne sont pas obligatoires : on peut par exemple n'avoir besoin de résoudre que 3 sous-problèmes de taille  $\frac{n}{4}$ .

Un exemple naturel de la technique *diviser pour régner* est l'algorithme de classement par fusion dans sa version *top-down* présenté ici.

### 14.2 Algorithme de Karatsuba

#### Problème

Il s'agit de multiplier deux nombres entiers, chacun représentés sur  $n$  bits où  $n$  peut dépasser la taille d'un mot mémoire. Si les deux multiplicandes  $p$  et  $q$  en base  $b$  s'écrivent :

$$p = \sum_{i=0}^n p_i b^i \quad q = \sum_{i=0}^n q_i b^i$$

alors la multiplication des deux nombres se ramène à la multiplication de polynômes :

$$p \times q = \sum_{k=0}^{2n} r_k b^k$$

où les coefficients valent :

$$r_k = \sum_{i+j} p_i q_j$$

Bien sûr, pour obtenir les « vrais » chiffres  $r_k$ , il ne faut pas oublier de faire une passe pour propager les retenues, sans quoi des chiffres pourraient dépasser  $b$ .

### Méthode par force brute

On peut bien évidemment programmer la méthode ci-dessus et obtenir un résultat correct. Une façon d'exprimer l'algorithme est la suivante :

```
function multiply(a[0..p], b[0..q], base)
    # Nouveau tableau pour le résultat
    product[0..p+q]
    for j <- 0 to q
        carry = 0
        for i <- 0 to p
            product[i + j] <- product[i + j] + carry + a[i] * b[j]
            carry = product[i + j] / base
            product[i + j - 1] <- product[i + j] % base
            product[j + p] += carry
    return product
```

Quelle est la complexité de cet algorithme ? Ici, la complexité va s'exprimer en fonction de la *longueur* en nombre de chiffres des nombres en entrée. Étant donné les deux boucles imbriquées, la complexité sera en  $O(p \times q)$  opérations arithmétiques élémentaires.

### Méthode diviser pour régner

En 1960, le mathématicien russe A. Karatsuba a remarqué que :

$$(a_1 \times 10^k + a_2)(b_1 \times 10^k + b_2) = a_1 b_1 \times 10^{2k} + (a_1 b_2 + a_2 b_1) \times 10^k + a_2 b_2$$

mais aussi :

$$(a_1 \times 10^k + a_2)(b_1 \times 10^k + b_2) = a_1 b_1 \times 10^{2k} + (a_1 b_1 + a_2 b_2 - (a_1 - a_2)(b_1 - b_2)) \times 10^k + a_2 b_2$$

Le gain consiste ici à remplacer 4 multiplications par seulement 3 multiplications. La 4<sup>ème</sup> multiplication est remplacée par des additions et soustractions à partir des autres produits. On peut calculer les produits intermédiaires récursivement.

La question se pose du choix de  $k$  : quelle valeur sera la plus efficace ? Si on coupe les nombres en deux parties égales à chaque fois, on aura le gain maximal. En effet, on déploiera un arbre binaire balancé.

Les détails de l'application de la méthode diviser pour régner au problème de la multiplication sont :

**Division**  $n \rightarrow \frac{n}{2}$ , détermination de  $a_1, a_2, b_1, b_2$

**Règne** calcul de  $a_1 b_1, a_2 b_2$  et  $(a_1 - a_2)(b_1 - b_2)$

**Combinaison** application de la formule de Karatsuba.

Si on suppose en entrée deux nombres à  $n$  bits, on peut mettre la complexité de l'algorithme en équation :

$$\begin{cases} T(1) = 1 \\ T(n) = 3T(\lceil \frac{n}{2} \rceil) + cn + d \end{cases}$$

Nous verrons ci-dessous que cette récurrence se résoud en :  $T(n) = \Theta(n^{\log_2 3})$ .

## 14.3 Analyse des algorithmes diviser pour régner

Considérons un problème que l'on résoud par la méthode diviser pour régner. Dans le cas général, on divise un problème initial de taille  $n$  en  $a$  sous-problèmes de taille  $n/b$ . Fréquemment, on utilise  $a = b = 2$  mais ce n'est pas toujours le cas.

La procédure générique pour résoudre notre problème s'écrit donc :

```
function solve(data)
    #c la taille de data est n
    if n < 1
        return data

    Divide(data)
    #c on obtient data_1, data_2, ... data_a
    #c a sous-ensembles de taille n/b

    #c on calcule les solutions partielles à
    #c chaque sous-problème

    s_1 = solve(data_1)
    s_2 = solve(data_2)
    ...répéter a fois...
    s_a = solve(data_a)

    #c il faut encore conquérir la solution globale au problème
    s = Conquer(s_1, s_2, ..., s_a)

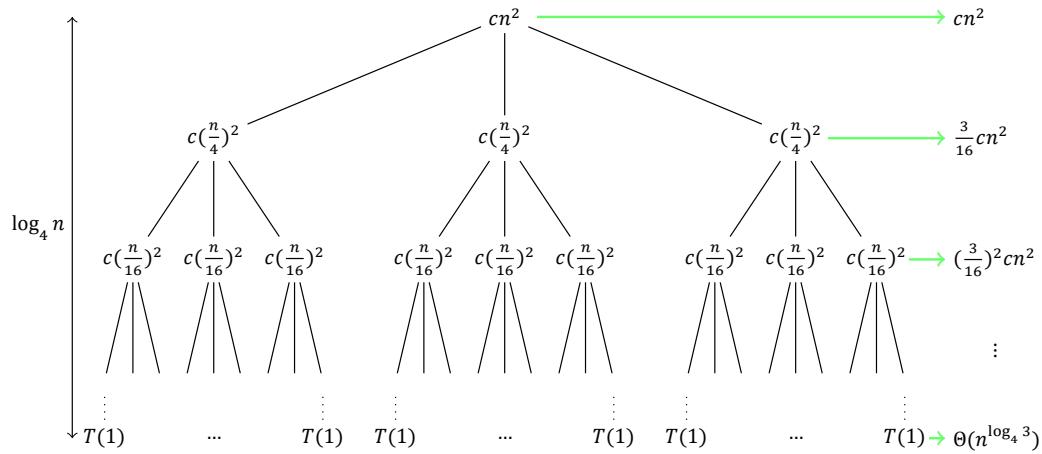
end procedure
```

Soit  $T(n)$  le temps d'exécution de cette fonction sur une donnée de taille  $n$ . Ce temps d'exécution vérifie la relation :

$$T(n) = 2T\left(\frac{n}{2}\right) + D(n) + C(n)$$

Le calcul de la complexité en temps d'une fonction de ce type nécessite la résolution d'une suite récurrente. Il est assez simple de trouver une relation pour  $T(n)$  en développant l'arbre des appels récursifs et en sommant les contributions de chaque noeud.

Exemple sur la relation particulière  $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$  ci-dessous.



$$T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$$

Pour calculer la complexité, il faut sommer les termes sur la colonne de droite :

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n - 1} \left( \frac{3}{16} \right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left( \frac{3}{16} \right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2).
\end{aligned}$$

## 14.4 Théorème maître

Ce théorème donne une borne asymptotique dans les cas les plus fréquents. Il se démontre en étudiant l'arbre développé précédemment dans chacun des cas. Une particularité importante de ce théorème : il permet d'éliminer les opérateurs  $\lfloor \rfloor$  et  $\lceil \rceil$  : ceux-ci sont absorbés dans les cas du théorème et les résultats sont valides même si la taille du problème d'entrée ne se divise pas exactement en  $a$  sous-problèmes de taille  $b$  à chaque pas.

Soit une relation de récurrence de la forme

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

avec  $a \geq 1, b > 1$  et  $f(n)$  une fonction asymptotiquement positive.

Soit  $a \geq 1$  et  $b > 1$  deux constantes,  $f(n)$  une fonction et soit  $T(n)$  définie sur les entiers positifs par la relation de récurrence  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$  où l'on peut prendre aussi bien  $\frac{n}{b}$ ,  $\lfloor \frac{n}{b} \rfloor$  que  $\lceil \frac{n}{b} \rceil$  au choix. Alors,  $T(n)$  peut être bornée asymptotiquement de la façon suivante :

- si  $f(n) = O(n^{\log_b a - \epsilon})$  pour une certaine constante  $\epsilon > 0$ , alors  $T(n) = \Theta(n^{\log_b a})$ ;
- si  $f(n) = \Theta(n^{\log_b a})$ , alors  $T(n) = \Theta(n^{\log_b a} \log_2 n)$ ;
- si  $f(n) = \Omega(n^{\log_b a + \epsilon})$  pour une certaine constante  $\epsilon > 0$  et si  $af(\frac{n}{b}) \leq cf(n)$  pour une certaine constante  $c < 1$  et tout  $n$  suffisamment grand, alors  $T(n) = \Theta(f(n))$ .

## 14.5 Exemples d'application du théorème maître

### Exemple 1

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

Ici,  $a = 9$ ,  $b = 3$ ,  $f(n) = n$  et  $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$ . Puisque  $\exists \epsilon > 0$   $f(n) = O(n^{\log_3 9 - \epsilon})$ , nous sommes dans le cas 1 du théorème et nous pouvons conclure que  $T(n) = \Theta(n^2)$ .

### Exemple 2

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

Ici,  $a = 1$ ,  $b = \frac{3}{2}$ ,  $f(n) = 1$  et  $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$ .

Le cas 2 du théorème s'applique, car  $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$ . Nous pouvons en conclure que la solution de la relation de récurrence est :

$$T(n) = \Theta(\log n).$$

### Exemple 3

$$T(n) = 3T\left(\frac{n}{4}\right) + n \log n$$

Ici,  $a = 3$ ,  $b = 4$ ,  $f(n) = n \log n$  et  $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$

Puisque  $\exists \epsilon > 0$   $f(n) = \Omega(n^{\log_4 3 + \epsilon})$ , le cas 3 du théorème s'appliquera si nous pouvons prouver la condition de régularité pour  $f(n)$ .

Pour  $n$  suffisamment grand et  $c = \frac{3}{4}$ ,  $af\left(\frac{n}{b}\right) = 3\left(\frac{n}{4}\right) \log\left(\frac{n}{4}\right) \leq \frac{3}{4}n \log n = cf(n)$ .

Donc le cas 3 du théorème s'applique et la solution de la relation de récurrence est :

$$T(n) = \Theta(n \log n).$$

### Exemple 4

$$T(n) = 2T\left(\frac{n}{2}\right) + n \log n$$

Le théorème maître ne s'applique pas à cette relation de récurrence. En effet, même si la relation a bien la forme appropriée :  $a = 2$ ,  $b = 2$ ,  $f(n) = n \log n$  et  $n^{\log_b a} = n$  le cas 3 du théorème ne

peut pas s'appliquer. On pourrait croire qu'il s'applique parce que  $f(n) = n \log n$  croît asymptotiquement plus vite que  $n^{\log_b a} = n$ . Le problème est que  $f(n)$  ne croît pas polynomialement plus vite que  $n$ . Le rapport  $\frac{f(n)}{n^{\log_b a}} = \frac{(n \log n)}{n} = \log n$  croît asymptotiquement moins vite que  $n^\epsilon$  pour tout  $\epsilon > 0$ .

De ce fait, cette relation de récurrence tombe entre les cas 2 et 3 du théorème.

On peut montrer une extension du cas 2 du théorème : si  $f(n) = \Theta(n^{\log_b a} \log^k n)$  avec  $k \geq 0$  alors  $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$ . Donc la solution à la relation de récurrence est :

$$T(n) = O(n \log^2 n).$$

## Programmation dynamique

### 15.1 Principe

La programmation dynamique est un schéma inventé par Bellman dans les années 50. L'idée de base consiste à observer que diviser pour régner ou tout autre schéma récursif peut être amené à recalculer des solutions à des sous-problèmes qui ont déjà été vus. Dans ce cas, il est utile d'établir un schéma qui permet la réutilisation automatique des solutions à ces sous-problèmes sans avoir à les recalculer.

Premier exemple de schéma récursif : fonction `fact()`

$$\begin{cases} \text{fact}(0) & = 1 \\ \text{fact}(n + 1) & = (n + 1) * \text{fact}(n) \end{cases}$$

Ici, aucun des appels récursifs n'est réutilisé.

Second exemple de schéma récursif : fonction `fib(n)`

$$\begin{cases} \text{fib}(0) & = 1 \\ \text{fib}(1) & = 1 \\ \text{fib}(n + 2) & = \text{fib}(n + 1) + \text{fib}(n) \end{cases}$$

Ici, le calcul de `fib(5)` fait appel aux calcul de `fib(4)` et de `fib(3)`, le calcul de `fib(4)` fait appel au calcul de `fib(3)` à nouveau.

Une programmation naïve de `fib(10)` conduit à calculer `fib(3)` 21 fois, et `fib(1)` 55 fois ! En fait, il faut  $\text{fib}(n - 1)$  appels à `fib(1)` pour calculer `fib(n)` !

Bien évidemment, l'utilisation d'une table va nous permettre de ne calculer chaque valeur intermédiaire qu'une seule fois.

Le schéma général de programmation dynamique se décompose en 4 étapes :

1. Caractériser la structure d'une solution optimale
2. Définir récursivement la valeur d'une solution optimale
3. Calculer la valeur d'une solution optimale, généralement de façon *ascendante (bottom-up)*
4. Construire une solution optimale à partir de l'information recueillie.

On peut omettre la dernière étape si on n'a besoin que de la *valeur* de la solution optimale – par opposition à la *solution* elle-même.

On dit qu'un problème à la propriété de *sous-structure optimale* si une solution optimale à ce sous-problème peut être déterminée à partir de solutions optimales à un certain nombre de ses sous-problèmes.

1. La solution au problème consiste en un choix initial. Ce choix laisse un ou plusieurs sous-problèmes à résoudre.
2. Pour un problème donné, vous recevez le choix qui conduit à une solution optimale. Vous ne cherchez pas à savoir comment atteindre ce choix.
3. Étant donné ce choix, vous déterminez quels sous-problèmes s'ensuivent et comment caractériser l'espace des sous-problèmes.
4. Vous montrez que les solutions aux sous-problèmes utilisées dans une solution optimale au problème, doivent être eux-mêmes optimaux en utilisant une technique de couper-coller (injecter une solution non-optimale à un sous-problème et la solution globale n'est plus optimale. Contradiction.).

## 15.2 Exemple 1 : le rendu de monnaie

Une première option possible se fonde sur l'équation :

$$C_s(v) = 1 + \min_{1 \leq i \leq n} C_s(v - c_i)$$

On peut interpréter cette équation de la façon suivante : le nombre de pièces optimal pour la valeur  $v$  par rapport au système  $s$ , c'est une pièce de plus que le nombre de pièces optimal pour la valeur  $v - c_i$  pour chacune des valeurs  $c_i$  de  $s$ .

La mise en place de ce schéma conduit au code suivant :

```
def min_pieces_dynamic_1(S, valeur):
    table = [0]*(valeur+1)
    for i in range(1, valeur+1):
        table[i] = 1 + min([table[i-S[k]] for k in range(len(S)) if i - S[k]
                           >= 0], default=0)
    return table[valeur]
```

La table est remplie avec les valeurs successives du nombre optimal de pièces pour une valeur qui correspond à l'indice. La complexité temporelle est en :  $O(V \times |S|)$  et la complexité spatiale en :  $O(V)$ .

On peut également utiliser une autre équation :

$$C(i, j) = \min(C(i - 1, j), 1 + C(i, j - s_i))$$

On peut interpréter cette équation de la façon suivante :  $C(i, j)$  désigne le nombre optimal de pièces pour la valeur  $j$  en utilisant uniquement les pièces de 1 à  $i$ . Pour rendre la valeur  $j$ , soit on utilise une pièce  $s_i$ , soit on ne l'utilise pas, et le nombre de pièces optimal est le minimum entre les deux.

La mise en place mot à mot de ce schéma aboutit au code suivant :

```

def min_pieces(S, i, valeur):
    if valeur == 0:
        return 0
    elif i == -1 or valeur < 0:
        return float('inf')
    else:
        return min(min_pieces(i-1, valeur), 1 + min_pieces(i, valeur-S[i]))
# Appel par min(S, len(S)-1, valeur) :
min_pieces([23, 7, 1], 2, 28)

```

Ici, pour éviter la redondance des calculs, on remplit une table à deux dimensions de taille  $n \times V$ .

```

def min_pieces_dynamic(S, valeur):
    m, n = len(S)+1, valeur+1
    table = [[0] * n for _ in range(m)]
    for j in range(1, n):
        table[0][j] = float('inf')
    for i in range(1, m):
        for j in range(1, n):
            if j - S[i-1] >= 0:
                v = table[i][j - S[i-1]]
            else:
                v = float('inf')
            table[i][j] = min(table[i-1][j], 1 + v)
    return table[m-1][n-1]

```

La complexité temporelle est en :  $O(V \times |S|)$  et la complexité spatiale en :  $O(V \times |S|)$ .

## 15.3 Exemple 2 : le sac à dos

### Le problème du sac à dos

Nous avons  $n$  fichiers de données à stocker, chaque fichier  $i$  possède une taille  $w_i$  et prend un temps  $t_i$  à recalculer. La capacité de stockage dont nous disposons est  $W$ . Nous cherchons à éviter au maximum à recalculer nos données. Nous devons donc trouver un sous-ensemble des fichiers à stocker tel que :

- la somme des tailles des fichiers soit inférieure à  $W$ ,
- le temps de calcul des fichiers stockés soit maximal.

Les fichiers sont insécables : c'est tout ou rien.

Comment sélectionner les fichiers ?

### Description formelle

Soit deux n-uplets d'entiers positifs :  $\langle v_1, v_2, \dots, v_n \rangle$  et  $\langle w_1, w_2, \dots, w_n \rangle$  et  $W > 0$ , déterminer le sous-ensemble  $T \subseteq \{1, 2, \dots, n\}$  tel que :

- $\sum_{i \in T} v_i$  soit maximal,
- $\sum_{i \in T} w_i \leq W$

Remarque : il existe plusieurs variantes de ce problème. Dans la variante décrite ici, chaque élément  $v_i$  ne peut être pris qu'une fois, on parle donc du problème du sac à dos *sans répétition*. On peut introduire une variante *avec répétition* où l'on a le droit d'utiliser le même  $v_i$  plusieurs fois.

Le problème du sac à dos est un problème d'optimisation. On peut le résoudre par la technique de la force brute, mais il faut essayer les  $2^n$  sous-ensembles de  $T$  ! Cette solution est rapidement impraticable bien entendu.

Peut-on faire mieux ?

### Solution par programmation dynamique

Nous nous intéressons à la valeur maximale que nous pouvons mettre dans le sac à dos :

$$\begin{aligned} K(w, j) &= \text{valeur maximale atteignable} \\ &\text{avec un sac-à-dos de capacité } w \text{ et les objets de } 1, \dots, j \end{aligned}$$

Nous cherchons  $K(W, n)$ .

La relation de récurrence qui régit notre problème est :

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}.$$

L'algorithme consiste à remplir une table à deux dimensions.

```

for j ← 0 to n
    K(0, j) = 0
for w ← 0 to W
    K(w, 0) = 0
for j ← 1 to n
    for w ← 1 to W
        if w[j] > w
            K[w, j] ← K[w, j - 1]
        else
            K[w, j] ← max(K[w, j - 1], K[w - w[j], j - 1] + v[j])
return K[W, n]

```

## Exploration en espace des états

### 16.1 Espace des états

Certains problèmes ne peuvent pas être résolus par une approche glouton ou diviser pour régner. Ces problèmes nécessitent d'explorer un graphe implicite de situations. C'est le cas pour des problèmes tels que les  $n$  reines, le voyageur de commerce, le taquin, le sudoku, la coloration de graphes, etc.

La modélisation de ces problèmes passe par la définition des *états* du problème et des objets suivants :

**état initial** l'état dans lequel se trouve le système au départ

**opérateurs** les actions élémentaires que l'on peut appliquer au système pour le faire changer d'état

**successeur** la fonction qui prend un état  $s$  et un opérateur  $op$  et calcule l'état  $s'$  résultant de l'application de  $op$  à  $s$

**espace d'états** le graphe des états définis par les états initiaux et la fonction successeur

**chemin** une séquence d'actions prises dans l'espace des états qui mène d'un état donné à un autre

**test de solution** fonction booléenne qui indique si un état donné est une solution à notre problème

**état final** état qui satisfait le test de solution

**coût d'un chemin** la somme des coûts des opérateurs sur le chemin

**solution** un chemin entre un état initial et un but.

Le graphe doit rester implicite : tous les problèmes que l'on attaque de cette façon ont un espace des états gigantesque. Il est impensable – et inutile – de développer le graphe des états a-priori.

### 16.2 Exemples de problèmes :

#### Problème des $n$ reines

Il s'agit de poser  $n$  reines sur un échiquier de taille  $n \times n$  sans que 2 reines quelconques soient en prise.

Chaque état décrit une situation de l'échiquier et la position des reines. L'état initial est un échiquier vide. Les opérateurs consistent à ajouter une nouvelle reine sur l'échiquier en respectant la contrainte que la nouvelle reine ne doit pas se trouver en prise. Le test de solution consiste à vérifier que  $n$  reines ont bien été posées. Ici, la solution au sens de chemin entre l'échiquier vide et l'échiquier rempli ne nous intéresse pas. Nous ne sommes intéressés que par la situation finale.

### Sudoku

Il s'agit de remplir une grille de Sudoku à partir d'une grille partiellement remplie.

Chaque état comprend la description de la grille avec les chiffres de chaque case. L'état initial est la grille partiellement remplie. Les opérateurs consistent à ajouter un chiffre dans une case vide en respectant les contraintes du Sudoku : chaque chiffre ne doit apparaître qu'une seule fois sur chaque ligne, sur chaque colonne et dans chaque sous-groupe. Le test de solution consiste à vérifier que la grille est effectivement remplie.

### Voyageur de commerce

Nous avons un graphe  $G = (V, E)$  de  $n = |V|$  villes reliées par des routes. Il s'agit de trouver un chemin qui passe par chacune des villes une seule fois et tel que la longueur de ce chemin soit minimale.

Ici, nous avons un problème d'optimisation. Nous pouvons nous reposer sur l'énumération des solutions au problème sous-jacent pour ensuite ne retenir que la meilleure d'entre elles.

Chaque état comprend la description du chemin en cours de construction. L'état initial comprend le chemin vide. Les opérateurs consistent à ajouter une arête au chemin partiel en vérifiant que nous ne passons pas deux fois par la même ville. Le test de solution consiste à vérifier que nous sommes bien passés par toutes les villes.

### Coloration de graphes

Soit un graphe  $G = (V, E)$ . Nous cherchons à affecter des couleurs prises dans un ensemble  $C$  à chaque noeud du graphe de façon à ce que deux noeuds adjacents n'aient pas la même couleur. De plus nous cherchons à minimiser la taille de  $C$ .

C'est à nouveau un problème d'optimisation. Nous pouvons énumérer les solutions au problème sous-jacent pour un  $C$  de taille donnée. Deux options s'offrent à nous :

- énumérer toutes les solutions et retenir la meilleure,
- nous arrêter dès qu'une solution est trouvée et recommencer avec un  $C'$  de taille inférieure, recommencer jusqu'à ce qu'aucune solution ne soit plus trouvée.

### Planification de chemin

Nous disposons d'un robot qui peut naviguer dans un espace discret à 2 dimensions. À chaque case, il peut choisir de se déplacer sur une des 8 cases voisines. L'espace est clos par des murs et il existe des cases occupées par des obstacles sur lesquelles le robot ne peut pas aller. Le robot doit se placer d'un point de départ à un point d'arrivée le plus rapidement possible.

L'état initial et l'état final sont les deux positions de départ et d'arrivée sur la grille. Les opérateurs consistent à se déplacer sur une case voisine qui ne soit pas interdites. Le test de solution consiste à vérifier que l'on est bien sur la position d'arrivée.

### 16.3 Performance

On mesurera la performance d'un algorithme d'exploration du graphe implicite des situations par le nombre de noeuds qu'il aura effectivement développés. Plus l'algorithme développera de noeuds, moins il sera performant. Si l'algorithme est capable d'aller à la solution sans développer aucun noeud superflu, il aura une performance maximale.

En général, on ramène cette performance au *facteur de branchement* d'un arbre uniforme correspondant à ce que l'algorithme de recherche d'une solution a développé. Si la solution trouvée possède une longueur  $d$  et que  $N$  noeuds ont été développés pour trouver cette solution, quel est le facteur de branchement d'un arbre uniforme qui possède  $N$  noeuds à la profondeur  $d$ ? Un arbre uniforme de facteur de branchement  $b$  développe  $b$  noeuds à la profondeur 1,  $b^2$  à la profondeur 2, ...  $b^d$  à la profondeur  $d$ . Le nombre de noeuds de cet arbre à la profondeur  $d$  est donc :

$$1 + b + b^2 + \dots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

On cherche donc  $b$  tel que :  $N = \frac{b^{d+1} - 1}{b - 1}$  et c'est ce paramètre qui va caractériser la performance de notre algorithme d'exploration de l'espace des solutions. La limite théorique idéale est  $b = 1$  : c'est le cas lorsqu'on sait aller droit au but et qu'on ne développe jamais de noeud superflu. Plus la valeur de  $b$  sera proche de 1, meilleur sera l'algorithme.

### 16.4 Backtracking

La technique du backtracking n'est jamais qu'une exploration en profondeur récursive du graphe implicite des états.

Le schéma général du backtracking s'exprime ainsi :

```
function backtrack(s, d)
    #c s est la situation courante
    #c d est la profondeur courante
    if isSolution(s)
        return True

    nextS ← computeNextSituations(s)
    for n in nextS
        if backtrack(n, d+1)
            return True

    #c aucune possibilité n'a marché
    return False
```

Nous construisons une solution de façon incrémentale, mais pas de façon monotone : contrairement au schéma glouton, nous acceptons de remettre en cause les choix précédents.

À chaque étape, plusieurs choix s'offrent à nous. Nous prenons un choix tout en mémorisant les autres. Nous développons ainsi une branche d'un arbre de possibilités. Si à un moment donné, nous tombons devant une impasse, ou bien que la solution construite n'est pas convenable, nous revenons en arrière au dernier point de choix et nous essayons les choix alternatifs. Chaque fois

que nous avons épuisé une liste de points de choix, nous revenons aux points de choix précédents. De cette façon, nous allons explorer exhaustivement toutes les possibilités. Soit nous trouverons notre solution, soit elle n'existe pas.

Le paramètre `d` de profondeur n'est pas obligatoire. Il peut nous aider à garder la trace de l'endroit où nous sommes.

Il faut faire attention à ne pas *boucler* dans le graphe des états : rien dans le pseudocode ne nous empêche de repasser plusieurs fois par le même état. Pour éviter de tomber dans une boucle, il est nécessaire de mettre en place une structure de donnée annexe comme un dictionnaire qui va mémoriser les états par lesquels nous sommes déjà passés. On peut alors retirer de `computeNextSituations(s)` les états que nous avons déjà rencontrés.

## 16.5 Backtracking : coloration de graphe

Voici une mise en oeuvre du backtracking sur le problème de la coloration de graphe.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 25 18:12:24 2015

@author: Fabrice
"""

def readGraph(filename):
    g = {}
    with open(filename) as f:
        for line in f.readlines():
            if line[0] == '#':
                continue
            nodes = [int(x) for x in line.split()]
            if not nodes[0] in g:
                g[nodes[0]] = {}
            for v in nodes[1:]:
                g[nodes[0]][v] = 1
    m = [[None for j in range(0, len(g)+1)] for i in range(0, len(g)+1)]
    for i in g:
        for j in g[i]:
            # print(i, j)
            m[i][j] = 1
    return m

# Vérification des contraintes au noeud k
# On vérifie chaque adjacent v à k :
# - k doit être différent de v
# - G[k][v] != None pour que v soit adjacent à k
# - c[v] doit être coloré
# - c[v] == c[k] avec la même couleur que k
def checkConstraints(G, c, k):
```

```

    for v in range(1, len(G)):
        if k != v and G[k][v] and c[v] and c[v] == c[k]:
            return False
    return True

# La fonction backtrackColor reçoit plusieurs paramètres :
# - G est le graphe réduit à sa matrice d'adjacence
# - c est une coloration initialement vide
# - k est le numéro du noeud courant
# - maxColors est le nombre de couleurs que l'on s'autorise
def backtrackColor(G, c, k, maxColors):
    # Condition d'arrêt
    if k == len(G):
        return c
    # On essaye toutes les couleurs pour le noeud k
    for color in range(1, maxColors+1):
        # assignation de la nouvelle couleur au noeud k
        c[k] = color
        # si les contraintes sont vérifiées
        if checkConstraints(G, c, k):
            # on essaye au niveau suivant
            res = backtrackColor(G, c, k+1, maxColors)
            # si ça a marché, on renvoie la valeur au
            # niveau précédent
            if res:
                return res
    # Echec, on efface la couleur pour le noeud k
    c[k] = None
    return None

G = readGraph('../graphs/graphe0.grf')

print(backtrackColor(G, [None for x in range(0, len(G))], 1, 6))

G = readGraph('../graphs/graphe1.grf')

print(backtrackColor(G, [None for x in range(0, len(G))], 1, 6))

# Graphe constraint 5-colorable
G = readGraph('../graphs/graphe3.grf')
# Solution trouvée rapidement
print(backtrackColor(G, [None for x in range(0, len(G))], 1, 6))
# Si on limite à 4 couleurs, il faut beaucoup plus de temps !
print(backtrackColor(G, [None for x in range(0, len(G))], 1, 4))

```

Ce code détermine uniquement une coloration avec moins que MAX\_COLORS. Si on veut une

coloration optimale en nombre de couleurs, il faut :

- soit poursuivre l'exploration en profondeur et retenir la meilleure coloration (solution peu efficace)
- soit relancer l'exploration avec un nombre de couleurs strictement inférieur de une couleur au nombre de couleurs utilisées par la coloration trouvée.

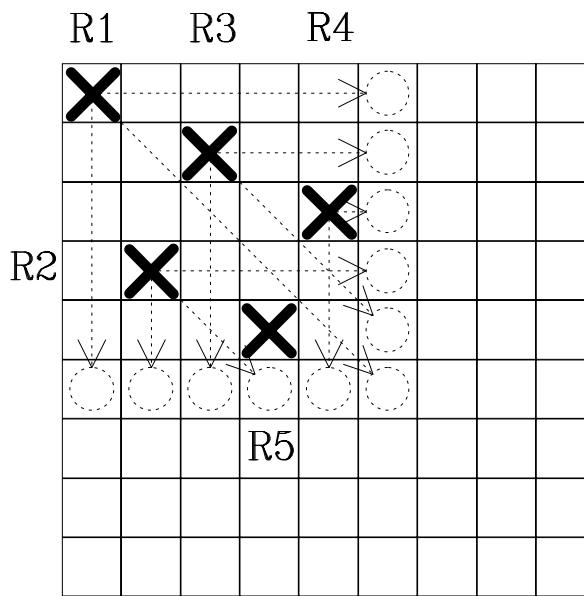
On peut encore trouver d'autres améliorations à cet exemple de code.

Trois exemples de graphes : [graphe1](#), [graphe2](#), [graphe3](#).

Les graphes sont lus comme des dictionnaires de dictionnaires, mais ensuite, on utilise leur matrice d'adjacence. Les noeuds sont donc implicitement numérotés de 1 à  $n$ .

## 16.6 Coloration de graphe : *forward checking*

L'heuristique de *forward checking* est très utile sur ce problème et modérément complexe à mettre en place. Cette heuristique consiste à constater que dans certaines situations, il ne sert à rien de revenir au point de choix strictement précédent : l'impossibilité constatée à l'état courant provient en fait de choix qui ont été effectués bien avant.



Ici, lorsqu'on cherche à poser la 6<sup>ème</sup> reine, on s'aperçoit que l'impossibilité remonte au choix de la 4<sup>ème</sup> reine et non au choix de la 5<sup>ème</sup> reine. Ceci aurait pu être détecté plus rapidement.

Il suffit d'associer un domaine des valeurs possibles à chaque variable du problème. Chaque fois que l'on fait un choix de couleur pour un noeud, on restreint les domaines des variables pour les noeuds adjacents. Si un des domaines devient vide, il n'y a pas de solution, on peut donc abandonner la branche en cours.

Voici une version du code modifié pour la coloration de graphe et qui met en oeuvre cette heuristique.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 27 09:09:36 2015
```

```

@author: Fabrice
"""

import copy

def readGraph(filename):
    g = {}
    with open(filename) as f:
        for line in f.readlines():
            if line[0] == '#':
                continue
            nodes = [int(x) for x in line.split()]
            if not nodes[0] in g:
                g[nodes[0]] = {}
            for v in nodes[1:]:
                g[nodes[0]][v] = 1
    m = [[None for j in range(0, len(g)+1)] for i in range(0, len(g)+1)]
    for i in g:
        for j in g[i]:
            # print(i, j)
            m[i][j] = 1
    return m

# Vérification des contraintes au noeud k
# On vérifie chaque adjacent v à k :
# - k doit être différent de v
# - G[k][v] != None pour que v soit adjacent à k
# - c[v] doit être coloré
# - c[v] == c[k] avec la même couleur que k
# - mise à jour du domaine de chaque variable :
#   - c[k] est enlevée du domaine de k
#   - c[k] est enlevée du domaine de chaque voisin de k
#   - si le domaine d'un voisin de k est vide, on renvoie None.

def checkConstraints(G, D, c, k):
    for v in range(1, len(G)):
        if k != v and G[k][v] and c[v] and c[v] == c[k]:
            return None
    newD = copy.deepcopy(D)
    newD[k].remove(c[k])
    for v in range(1, len(G)):
        if k != v and G[k][v]:
            if c[k] in newD[v]:
                newD[v].remove(c[k])
            if len(newD[v]) == 0:
                # print('No more options at ', k, v)
                return None

```

```

    return newD

# La fonction backtrackColor reçoit plusieurs paramètres :
# - G est le graphe réduit à sa matrice d'adjacence
# - D garde la liste des couleurs possibles pour chaque noeud
# - c est une coloration initialement vide
# - k est le numéro du noeud courant
# - maxColors est le nombre de couleurs que l'on s'autorise
def backtrackFCCColor(G, D, c, k, maxColors):
    # Condition d'arrêt
    if k == len(G):
        return c
    # On essaye toutes les couleurs pour le noeud k
    for color in D[k]:
        # assignation de la nouvelle couleur au noeud k
        c[k] = color
        # si les contraintes sont vérifiées
        newD = checkConstraints(G, D, c, k)
        if newD:
            # on essaye au niveau suivant
            res = backtrackFCCColor(G, newD, c, k+1, maxColors)
            # si ça a marché, on renvoie la valeur au
            # niveau précédent
            if res:
                return res
    # Echec, on efface la couleur pour le noeud k
    c[k] = None
    return None

# G = readGraph('../graphs/graphe0.grf')

# print(backtrackFCCColor(G, [None for x in range(0, len(G))], 1, 6))

# G = readGraph('../graphs/graphe1.grf')

# print(backtrackFCCColor(G, [None for x in range(0, len(G))], 1, 6))

# Graphe constraint 5-colorable
G = readGraph('../graphs/graphe3.grf')
# Solution trouvée rapidement
# print(backtrackFCCColor(G, [None for x in range(0, len(G))], 1, 6))
# Si on limite à 4 couleurs, il faut beaucoup plus de temps !
maxColors = 4
colorSet = set()
for c in range(1, maxColors+1):
    colorSet.add(c)

```

```
print(backtrackFCCColor(G, [copy.deepcopy(colorSet) for x in range(0, len(G))
    ], [None for x in range(0, len(G))], 1, 4))
```

## 16.7 Exploration générique

Le backtracking est une technique très simple d'exploration de l'espace des états. On peut établir un schéma plus générique de cette exploration qui supporte non seulement l'exploration en profondeur par backtracking mais aussi beaucoup d'autres alternatives comme l'exploration en largeur, ou encore l'algorithme  $A^*$  décrit ci-dessous.

Au lieu d'utiliser directement les états du problème, nous allons mettre en place une structure de noeud un peu plus informative. Chaque noeud comprendra 5 champs :

1. la description de l'état auquel le noeud correspond
2. le noeud parent
3. l'opérateur qui a été utilisé pour générer ce noeud
4. la profondeur à laquelle se situe ce noeud
5. le coût du chemin (minimal) entre le noeud de départ et ce noeud.

La fonction `Expanse()` est responsable de la détermination des noeuds successeurs d'un noeud donné et elle calcule chacune des composantes des noeuds qu'elle génère.

Un algorithme générique d'exploration de l'espace des états peut s'exprimer ainsi :

```
ExplorationGenerique(probleme, EnfileFN)
    noeuds ← CreeFile(CreeNoeud(EtatInitial(probleme)))
    while True
        if Vide?(noeuds)
            return echec
        noeud ← Defile(noeuds)
        si TestSolution(probleme)(Etat(noeud))
            return noeud
        noeuds ← EnfileFN(noeuds, Expanse(noeud, Operateurs(probleme)))
    end
```

Ceci suppose bien sûr une structure ad-hoc à la fois pour la représentation des noeuds et pour la représentation des problèmes :

- `EtatInitial()` prend un problème et renvoie son état initial
- `CreeNoeud()` fabrique un noeud à partir d'un état
- `CreeFile()` fabrique une file (pile, file, file de priorité) initialisée avec un noeud donné
- `Defile()` sort le premier noeud de la structure de file (pile, file, file de priorité)
- `Vide?()` teste si la file est vide
- `TestSolution()` prend un problème et renvoie la fonction de test de solution, cette fonction prenant elle-même un état et renvoyant un booléen indiquant si cet état est solution ou non du problème
- `EnfileFN()` ajoute des noeuds à la file, selon le type de structure choisie : au début pour une pile, à la fin pour une file, en fonction des priorités pour une file de priorité

- `Expanse()` calcule les noeuds successeurs d'un noeud donné, en fonction des opérateurs applicables
- `Operateurs()` retrouve les opérateurs du problème.

Cette formulation générique de l'exploration à l'avantage de permettre de regrouper sous une même structure différentes stratégies d'exploration.

Les noeuds qui sont dans la file sont dits être sur la *frontière* de l'espace exploré. On les qualifie aussi de noeuds *ouverts*. Les noeuds qui sont sortis de la file sont dits *clos*.

## 16.8 Exploration en profondeur

Si nous utilisons `ExplorationGenerique()` en choisissant une pile pour la file et la fonction `push()` pour la fonction `EnfileFN()`, nous obtenons une exploration en profondeur du graphe. Cette exploration est très similaire à l'algorithme par backtracking. Il faut faire attention aux mêmes problèmes : ne pas remettre dans la file de noeuds, des noeuds que nous aurions pu déjà rencontrer.

## 16.9 Exploration en largeur

Si nous utilisons `ExplorationGenerique()` en choisissant effectivement une file pour la file et la fonction `enqueue()` pour la fonction `EnfileFN()`, nous obtenons une exploration en largeur du graphe. Cette exploration est souvent impraticable : la file de noeuds grossit de façon démesurée, puisque si chaque noeud possède  $b$  successeurs, la file stockera  $b$  noeuds, puis  $b^2$ , puis  $b^3$  etc et si la solution se trouve à une profondeur  $d$ , il aura fallu stocker  $b^d$  noeuds dans la file.

## 16.10 Exploration à coût uniforme ou UCS

Ici, nous allons utiliser l'exploration générique avec une file de priorité : les noeuds y seront rangés en fonction de leur coût par rapport au noeud initial. De cette façon, lors de l'exploration, les noeuds seront développés en fonction de leur coût, et chaque fois qu'un noeud sera développé, ce sera parce qu'il sera de coût minimal. Lorsque la solution sera trouvée, on aura donc une solution à coût minimal.

L'inconvénient de l'exploration à coût uniforme réside dans le fait qu'on n'utilise aucune information sur la *direction* dans laquelle se trouve la solution.

## 16.11 Exploration *Best-First*

Si on dispose d'une information qui qualifie le coût d'un chemin entre un noeud donné et une solution, on peut avoir envie d'aller « droit au but » et choisir systématiquement le noeud dont on pense qu'il est le plus proche de la solution. Bien sûr, une telle évaluation n'est jamais certaine. Mais si on dispose d'une fonction *heuristique*  $h()$  qui possède les propriétés suivantes :

- $h(s) = 0$  si  $s$  est une solution
- $h(s) > 0$  si  $s$  n'est pas une solution

on peut utiliser l'exploration générique avec une file de priorité où les noeuds sont classés en utilisant cette fonction. L'exploration *Best-First* ne garantit absolument pas l'optimalité de la solution trouvée. Elle ne garantit même pas de trouver une solution.

Dans le cas du problème de planification de chemin, on peut malgré tout observer son efficacité en terme de noeuds développés : le chemin calculé n'est pas optimal, mais il est déterminé très rapidement.

## 16.12 Exploration A\*

L'exploration A\* résulte de la combinaison de l'exploration à coût uniforme avec l'exploration Best-First en ajoutant de bonnes conditions sur la fonction heuristique.

En effet, l'exploration à coût uniforme développe une frontière qui s'étend de façon uniforme autour du noeud de départ. Si on regarde la frontière à différentes étapes de l'algorithme, on voit des contours successifs, centrés sur le noeud de départ, et circulaires.

L'idée de A\* consiste à introduire l'information heuristique pour *déformer* ces contours en direction de la solution, et ainsi arriver à la solution en développant moins de noeuds que par UCS.

Pour continuer à garantir l'optimalité de la solution trouvée, il faut imposer à la fonction heuristique d'être *admissible* :

- $h(s) = 0$  si  $s$  est une solution
- $h^*(s) \geq h(s) > 0$  si  $s$  n'est pas une solution

La valeur  $h^*(s)$  est le coût minimal exact d'un chemin entre  $s$  et une solution. Ce coût est bien sûr inconnu, sinon le problème serait résolu. La fonction  $h()$  doit être admissible revient à dire qu'elle ne doit jamais surestimer le coût d'un chemin entre un noeud et une solution.

Dans le cas de la planification de chemin, il est bien évident que la distance euclidienne constitue une heuristique admissible, puisqu'il est impossible d'aller d'un noeud à un autre plus rapidement qu'en ligne droite.

On peut prouver que si l'on dispose de deux heuristiques  $h_1()$  et  $h_2()$  telles que  $h^*(s) \geq h_2(s) \geq h_1(s) > 0$ , alors la version de A\* utilisant  $h_2$  développe moins de noeuds que celle utilisant  $h_1$ . On dit que la version utilisant  $h_2$  est *plus informée* que celle utilisant  $h_1$ .



## Méta-heuristiques

### 17.1 Méta ... quoi?

Lorsque l'exploration en espace des états est impraticable, il faut se rabattre sur des techniques plus rapides et espérer qu'elles convergent malgré tout vers une solution.

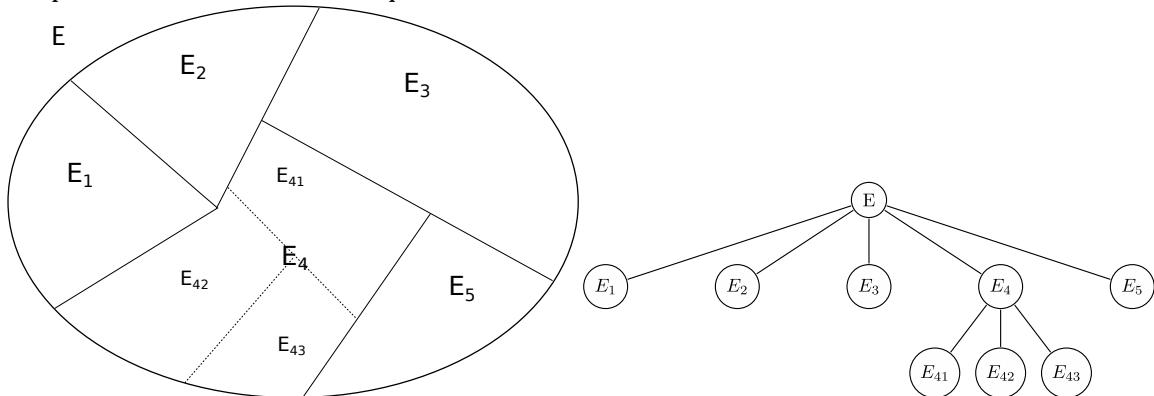
L'objectif ici est uniquement d'introduire l'existence de quelques techniques qui peuvent se révéler très puissantes, mais qui n'offrent pas de garantie de solution ou d'optimalité de la solution trouvée.

### 17.2 Branch and Bound

Cette technique consiste à *élaguer* les branches de l'arbre développé lors d'une exploration en profondeur (backtracking) de l'espace des solutions. On utilise une information sur les bornes inférieures et supérieures de l'intervalle dans lequel peut se trouver une solution optimale pour couper des branches lors de l'exploration<sup>1</sup>.

Le principe de cette technique consiste à :

- séparer (*branch*) l'ensemble des solutions en sous-ensembles plus petits
- explorer ces sous-ensembles en utilisant une évaluation optimiste pour majorer (*bound*) les sous-ensembles et ne plus considérer que ceux susceptibles de contenir une solution potentiellement meilleure que la meilleure solution courante.



1. Cette technique est à rapprocher de la technique  $\alpha - \beta$  utilisée pour élaguer un arbre minimax dans le calcul d'un coup pour un jeu à deux joueurs.

Soit  $E$  l'ensemble des solutions au problème, supposé discret, fini mais très grand. Nous énumérons les éléments de  $E$ , mais en les séparant en sous-ensembles non-vides, pas obligatoirement disjoints (généralement ils le sont). Nous itérons le processus jusqu'à des sous-ensembles de taille unitaire.

Sur la figure :  $E$  est séparé en 5 sous-ensembles, le sous ensemble  $E_4$  est séparé en 3 sous-ensembles, etc.

Les feuilles de l'arbre sont les solutions potentielles au problème. Nous cherchons la solution optimale, donc nous cherchons à nous diriger dans l'arbre vers la (une) solution optimale le plus rapidement possible.

Le schéma générique de cet algorithme est le suivant :

```
function solveBB(problem)
    #c active nodes
    AN ← getAN(problem)
    #c initialisation de la
    #c meilleure solution
    best ← nil
    while AN ≠ ∅
        n ← chooseAN(AN)
        if solution(n)
            if n > best
                best ← n
            else
                children ← split(n)
                for f in children
                    e ← eval(f)
                    if e > best
                        AN ← AN ∪ {f}
    return best
```

avec les définitions suivantes :

**Active nodes** liste  $AN$ , équivalent de la frontière dans l'exploration générique vu plus haut

**Séparation** division du noeud  $n$  en ses fils  $children$  par la fonction  $split$ .

**Évaluation optimiste** les feuilles sont solution, et ont une valeur exacte. Les noeuds internes ne sont pas des solutions complètes, ils sont **évalués** par un majorant de toutes les solutions contenues dans le sous-arbre démarrant à ce noeud. Aspect crucial : il faut majorer au plus près !

**Élagage** interruption de l'exploration. Inutile de diviser le noeud lorsque :

- on atteint une feuille
- l'évaluation du noeud courant n'est pas meilleure que la meilleure solution trouvée.

### 17.3 Branch and bound : sac à dos

Nous appliquons la technique Branch and Bound à un cas concret du problème du sac à dos :

- $W = 130$
- 4 items

	item 1	item 2	item 3	item 4
coût	4	5	6	2
poids	33	49	60	32

Il faut donc maximiser :

$$4x_1 + 5x_2 + 6x_3 + 2x_4$$

sous les contraintes que les  $x_i$  sont des entiers positifs et

$$33x_1 + 49x_2 + 60x_3 + 32x_4 \leq 130.$$

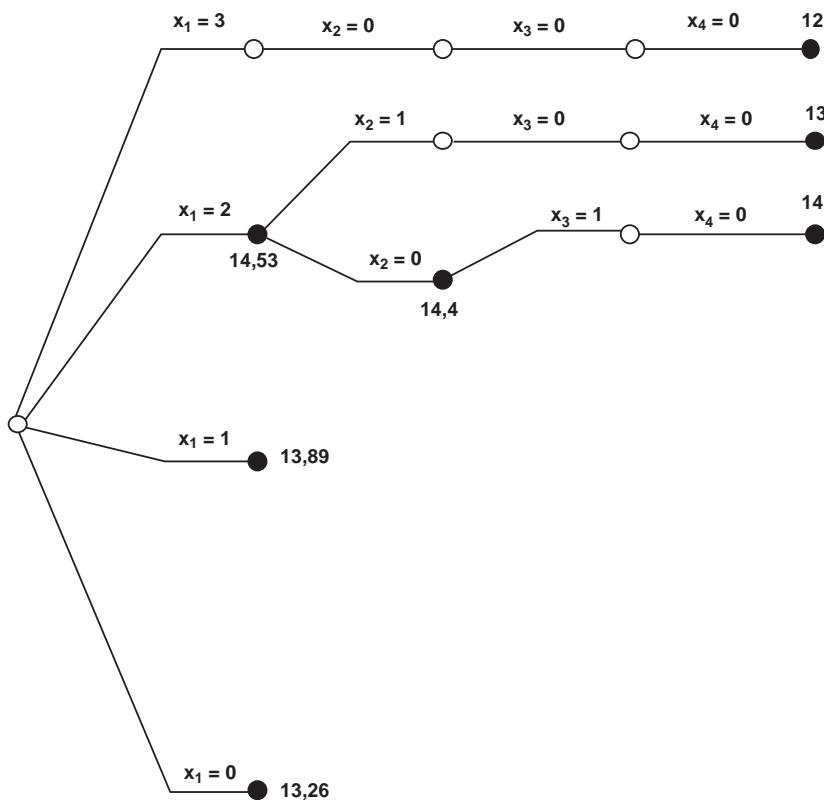
L'ensemble  $E$  est celui de tous les quadruplets  $(x_1, x_2, x_3, x_4)$  d'entiers positifs.

Nous cherchons une solution optimale. Dans le cas où plusieurs solutions optimales existent, leur énumération prendrait un temps considérable !

Ici, nous adoptons les choix suivants :

**Séparation** : celle illustrée précédemment selon  $x_1$ , puis  $x_2$ , puis  $x_3$  puis  $x_4$ .

**Évaluation** : une évaluation optimiste simple consiste à remplir le reste du volume du sac par l'item qui rapporte le plus par unités de volume. On considère donc une évaluation sur  $x_i$  non entier. Majorant très grossier.



- $x_1 = 3$  : on ne peut plus mettre aucun item dans le sac, on a une meilleure solution avec 12.
- $x_1 = 2$  : item 2 présente le meilleur coût / volume. Noeud évalué à  $\$ 2+5/49(130-2 \times 33) = 14.53\$$ . L'évaluation est potentiellement supérieure à  $12 + 1 = 13$  donc le noeud est séparé.

- $x_1 = 2, x_2 = 1$  : on ne peut plus mettre aucun item dans le sac, on a une nouvelle meilleure solution avec 13
- $x_1 = 2, x_2 = 0$  : item 3 présente le meilleur coût/volume. Noeud évalué à  $2 \times 4 + 0 + 6/60(130 - 2 \times 33) = 14.4$ . L'évaluation est potentiellement supérieure à  $12 + 1 = 13$  donc le noeud est séparé.
- $x_1 = 2, x_2 = 0, x_3 = 1$  : on ne peut plus mettre aucun item dans le sac, on a une nouvelle meilleure solution avec 14.
- $x_1 = 1$  : évaluation de  $4 + 5/49(130 - 33) = 13.89$ , élagué.
- $x_1 = 0$  : évaluation de 13.26, élagué.

# **Quatrième partie**

# **Algorithmes**



## Classement

### 18.1 Définition formelle du problème du classement

Le problème du classement des nombres entiers peut être défini formellement de la façon suivante :

- Entrée : une liste d'entiers  $(a_1, a_2, \dots, a_n)$
- Sortie : une permutation  $(a'_1, a'_2, \dots, a'_n)$  des entiers en entrée
- Relation :  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

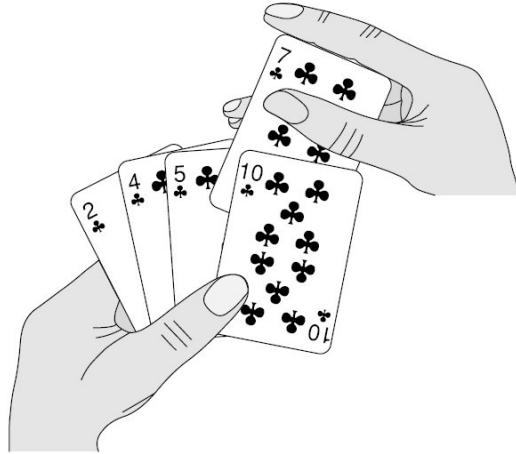
On peut définir de la même façon le problème général du classement d'une liste d'objets. Il faut supposer que chaque objet  $a_i$  possède un attribut spécifique  $a_i.key$  que l'on nomme sa *clé* et qu'il existe une relation d'ordre sur l'ensemble des clés des objets.

Dans le cas des entiers, la clé est l'objet lui-même. Si on veut classer une liste d'étudiants, on peut le faire par :

- ordre alphabétique patronymique
- ordre des codes INE
- ordre des numéros INSEE
- etc.

### 18.2 Classement par insertion

La technique de classement par insertion est celle que l'on utilise naturellement lorsqu'on veut ranger les cartes que l'on vient de nous distribuer et que l'on tient dans une main.



Cette technique de classement utilise une méthode glouton : la solution au problème est construite de manière incrémentale et monotone.

La preuve de l'algorithme repose sur l'utilisation d'un invariant de boucle : l'invariant de la boucle extérieure (ligne 2) est : le sous-tableau  $A[0..j-1]$  est classé. Il est facile de prouver que :

1. c'est le cas à l'initialisation parce que  $j$  vaut 1 et que le tableau  $A[0..0]$  est trivialement classé
2. cette condition est préservée après chaque itération : la boucle while (ligne 6) a pour rôle de décaler les éléments vers la droite pour faire de la place pour l'élément  $A[j]$ .
3. cette boucle termine car  $j$  augmente de façon monotone et finit donc par dépasser la valeur  $n-1$ . À ce moment l'invariant est : le sous-tableau  $A[0..n-1]$  est classé, le tableau est donc classé en entier.

Bien sûr, ceci repose sur le fait que la boucle while (ligne 6) est correcte : il faudrait la prouver à l'aide d'un autre invariant de boucle.

Le calcul de complexité se déroule comme suit :

---

#### Fonction InsertionSort(A)

---

	▷ coût	nombre
1 <b>for</b> $j \leftarrow 1$ <b>to</b> $A.length - 1$ <b>do</b>	$c_1$	$n$
2     key $\leftarrow A[j]$	$c_2$	$n - 1$
$\triangleright$ Insérer $A[j]$ dans		
la séquence classée $A[0..j-1]$		
3     $i \leftarrow j - 1$	$c_4$	$n - 1$
4     <b>while</b> ( $i \geq 0$ ) and ( $A[i] > key$ ) <b>do</b>	$c_5$	$\sum_{j=1}^{n-1} t_j$
5         $A[i+1] \leftarrow A[i]$	$c_6$	$\sum_{j=1}^{n-1} (t_j - 1)$
6         $i \leftarrow i - 1$	$c_7$	$\sum_{j=1}^{n-1} (t_j - 1)$
7     <b>end</b>		
8     $A[i+1] \leftarrow key$	$c_9$	$n - 1$
9 <b>end</b>		

---

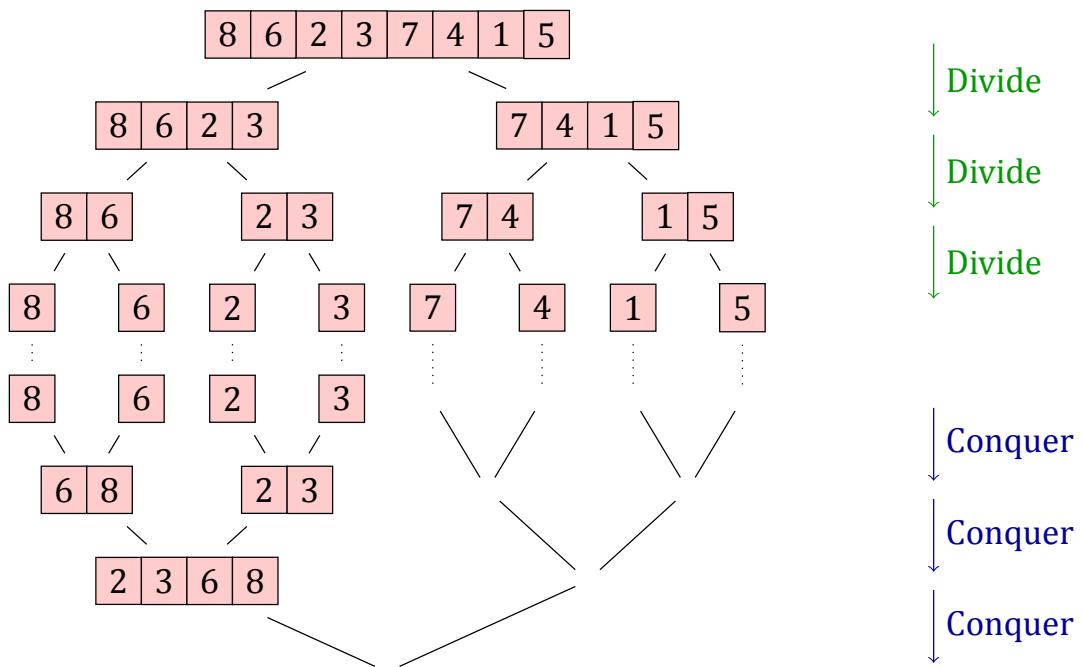
Ici,  $t_j$  désigne le nombre de fois que le test de la boucle while à la ligne 5 aura été entrepris pour cette valeur de  $j$ . On en déduit le coût total :

$$\begin{aligned}
 T(n) = & c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=1}^{n-1} t_j \\
 & + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7 \sum_{j=1}^{n-1} (t_j - 1) + c_9(n - 1)
 \end{aligned}$$

À partir de cette formule, on peut dériver la complexité de l'algorithme dans le meilleur cas, dans le pire cas, et dans le cas moyen *sous hypothèse d'une distribution aléatoire uniforme* de la permutation d'origine.

### 18.3 Classement par fusion

On peut illustrer le principe général de la méthode par le schéma suivant :



Le classement par fusion repose sur une opération de fusion de deux tableaux classés. Le processus consiste à scinder le tableau en deux moitiés de longueur égale que l'on classe récursivement par le même algorithme. Lorsqu'on arrive à des tableaux de longueur nulle ou de longueur 1, ceux-ci sont trivialement classés. Lorsqu'on a classé deux sous-tableaux, il faut les fusionner pour obtenir un tableau classé. Cette fusion passe par l'utilisation de tableaux auxiliaires. On recopie les éléments des sous-tableaux dans deux tableaux auxiliaires. Les tableaux auxiliaires sont ensuite parcourus simultanément. On choisit à chaque fois le plus petit élément que l'on peut trouver et on le copie dans le tableau d'origine.

Le classement global va opérer de la façon suivante :

Le pseudo-code de la fonction de classement est le suivant :

```
function mergeSort(A, p, r)
    if (r > p)
        #c Division en 2 sous-problèmes
        q ← (p+r)/2
        #c Résolution récursive des 2 sous-problèmes
        mergeSort(A, p, q)
        mergeSort(A, q+1, r)
        #c Conquête en fusionnant les 2 solutions
        merge(A, p, q, r)
    end

```

Le classement repose sur l'opération de fusion dont le pseudo-code est :

```

#c les tableaux left et right existent
#c et sont de taille au moins égale à celle de A
function merge(A, p, q, r)
    n1 ← q - p + 1
    n2 ← r - q
    for i ← 0 to n1-1
        left[i] ← A[p+i]
    for j ← 0 to n2-1
        right[j] ← A[q+1+j]
    i ← 0
    j ← 0
    for k ← p to r
        if j = n2 || left[i] < right(j)
            A[k] ← left[i]
            i ← i+1
        else
            A[k] ← right[j]
            j ← j+1
    end

```

On remarque qu'on réutilise les deux tableaux `left` et `right` : bien qu'ils soient utilisés récursivement dans la procédure de fusion, rien ne nous oblige à allouer de nouveau tableaux à chaque étape. Il y a ici une économie considérable à réutiliser ces deux tableaux.

Cette technique de classement utilise une méthode diviser pour régner appliquée de bas en haut ou *bottom-up* : les données sont d'abord scindées et ne sont classées que lorsque la fusion opère, c'est-à-dire en remontant de la récursion.

La complexité de l'opération de fusion est en  $O(n)$  où  $n$  est le nombre d'éléments à fusionner. On remarquera qu'on ne peut pas descendre en dessous de  $n - 1$  comparaisons, parce que sinon, un élément ne serait comparé à aucun autre. L'algorithme de fusion ne serait donc pas correct puisque que cet élément encourrait le risque de ne pas être correctement classé.

La complexité de l'algorithme de classement par fusion suit la loi  $T(n) = 2T(\frac{n}{2}) + cn$  puisqu'on divise chaque ensemble à classer en deux moitiés de longueur identique.

La complexité du classement par fusion est donc en  $O(n \log n)$  en vertu du théorème maître (par exemple).

Note : il existe d'autres implémentations possibles du classement par fusion et le processus de fusion peut être appliqué *top-down* ou *bottom-up*. Voir [https://en.wikipedia.org/wiki/Merge\\_sort](https://en.wikipedia.org/wiki/Merge_sort).

## 18.4 Classement rapide

Le classement rapide utilise également la méthode diviser pour régner, mais appliquée de haut en bas ou *top-down* : les données sont séparées entre celles qui sont plus petites qu'un pivot choisi arbitrairement et celles qui sont plus grandes.

La fonction `partition` s'occupe de séparer les données par rapport au pivot. Il faut bien noter que les données de part et d'autre du pivot ne sont pas classées.

```

function partition(A, p, r)
    x ← A[r]
    i ← p-1

```

```

for j ← p to r-1
  if A[j] <= x
    i ← i+1
    Swap(A[i], A[j])
  Swap(A[i+1], A[r])
  return i+1
end

```

L'algorithme de classement rapide n'a plus qu'à appliquer récursivement le processus de séparation jusqu'à tomber sur des tableaux à 0 ou 1 éléments qui sont bien évidemment classés par essence.

```

function quicksort(A, p, r)
  if p < r
    q ← partition(A, p, r)
    quicksort(A, p, q - 1)
    quicksort(A, q + 1, r)
end

```

## 18.5 Classement par tas

Le classement par tas repose sur la structure de tas vue comme une file de priorité, donc un tas maximal. La file de priorité réalise un classement partiel des éléments : on sait où se trouve le plus grand élément, et on sait l'extraire tout en restaurant la propriété de tas maximal rapidement. Si on applique ce processus de façon répétée, on peut sortir les éléments du tas par ordre décroissant.

```

function heapSort(A)
  BuildMaxHeap(A)
  heapSize ← A.length
  for i ← heapSize-1 downto 1
    swap(A, 0, i)
    heapSize ← heapSize-1
    MaxHeapify(A, 0)
end

```

Les données à classer sont tout d'abord arrangées en un tas. Cette opération est prise en charge par `BuildMaxHeap` avec un coût de  $O(n)$ . Ensuite de quoi les éléments sont extraits du tas (opération `ExtractMax` pour un coût  $O(\log n)$ ) un par un pour un coût total  $O(n \log n)$ .

## 18.6 En pratique

Classement	Meilleur cas	Pire cas	Cas moyen	Stabilité	Espace
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	oui	en place
Fusion	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	oui	$2n$
Tas	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	non	en place
Rapide	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	possible	en place

Un algorithme de classement est qualifié de *stable* si

$$(1 \leq i < j \leq n) \wedge (A[i].key = A[j].key) \Rightarrow (p(i) < p(j))$$

où  $p()$  désigne la permutation qui réalise le classement, i.e.  $p(i)$  est la position de  $A[i]$  après le classement du tableau.

# Graphes

## 19.1 Parcours

### Motivation

Lorsqu'on est en présence d'un graphe, l'opération la plus immédiate que l'on puisse vouloir réaliser consiste à visiter chaque noeud du graphe. Par exemple, le parcours peut avoir comme objectif d'imprimer ou de collecter une certaine information concernant chaque noeud.

Il peut y avoir des contraintes plus restrictives sur le parcours comme par exemple passer par chaque arête une seule fois (cycle eulérien) ou passer par chaque noeud une seule fois (chemin hamiltonien). Nous ne nous intéresserons pas à ces contraintes pour l'instant.

La définition d'un algorithme de parcours suppose :

- la donnée d'un noeud source ou noeud de départ, imposé ou choisi aléatoirement,
- la donnée d'une stratégie ou d'une politique de choix du noeud suivant parmi l'ensemble des noeuds adjacents au noeud courant

Cette stratégie doit bien entendu être fixée à l'avance. Il existe deux stratégies naturellement très simples : le parcours en largeur et le parcours en profondeur.

### Parcours en largeur

Le parcours en largeur repose sur la politique suivante : on part d'un noeud source, on examine tous les noeuds à une arête de la source, puis tous ceux à deux arêtes, etc. Cette politique consiste à s'éloigner le moins possible du noeud source. On ne passe aux noeuds à  $n + 1$  arêtes de la source que lorsqu'on a examiné tous ceux qui sont à  $n$  arêtes de la source.

```
function BreadthFirstSearch(G, s)
    for u ∈ V(G) - {s}
        u.color ← WHITE
        u.d ← ∞
        u.parent ← NIL
    s.color ← RED
    s.d ← 0
    s.parent ← NIL
    Q ← ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅
        u ← DEQUEUE(Q)
```

```

for each  $v \in \text{Adj}[u]$ 
  if  $v.\text{color} = \text{WHITE}$ 
     $v.\text{color} \leftarrow \text{RED}$ 
     $v.d \leftarrow u.d + 1$ 
     $v.parent \leftarrow u$ 
     $\text{ENQUEUE}(Q, v)$ 
     $u.\text{color} \leftarrow \text{GREEN}$ 
  end

```

### Complexité du parcours en largeur

Après l'initialisation, aucun noeud n'est remis à BLANC. Donc tous les noeuds sont mis dans la file une seule fois. Ils sont donc sortis de la file une seule fois également.

Les opérations `Enqueue()` et `Dequeue()` prennent un temps constant  $O(1)$  soit un total de  $O(V)$ .

La liste d'adjacence de chaque noeud est parcourue une seule fois lorsque le noeud est sorti de la file. La somme des longueurs des listes d'adjacence est en  $\Theta(E)$ , donc le temps total pour parcourir les listes d'adjacence est en  $\Theta(E)$ .

Le temps total pour parcourir le graphe en largeur est  $O(|V| + |E|)$ .

### Propriétés du parcours en largeur

Cet algorithme a la particularité de développer un arbre construit par la relation parent. Cet arbre permet de remonter de chaque noeud du graphe vers le noeud source en suivant un chemin unique qui a été calculé par le parcours en largeur. Ce chemin possède la propriété d'être un plus court chemin *en nombre d'arêtes* vers la source.

Soit un graphe  $G = (V, E)$  qui peut être orienté ou non et  $s \in V$  un noeud source arbitraire.

La distance *plus court chemin*  $\delta(s, v)$  entre les noeuds  $s$  et  $v$  est la longueur d'un chemin de longueur minimale *en nombre d'arêtes* parmi l'ensemble des chemins qui relient  $s$  à  $v$ , ou  $\infty$  si il n'y a pas de chemin.

Pour toute arête  $(u, v) \in E$ , nous avons  $\delta(s, v) \leq \delta(s, u) + 1$ .

**Théorème 19.1.** Soit  $G = (V, E)$  sur lequel on a executé BFS à partir de  $s \in V$ . Alors pendant son exécution, BFS atteint tous les noeuds  $v \in V$  qui sont atteignables depuis  $s$ . À la fin,  $v.d = \delta(s, v)$  pour tout  $v \in V$ . De plus, pour tout noeud  $v \neq s$  atteignable depuis  $s$ ; l'un des plus courts chemins en nombre d'arêtes de  $s$  à  $v$  est un plus court chemin de  $s$  à  $v.parent$  suivi de l'arête  $(v.parent, v)$ .

On commence par prouver par induction sur le nombre d'opérations `Enqueue()` qu'après terminaison de l'algorithme  $\forall v \in V \quad v.dist \geq \delta(s, v)$ . Ensuite démonstration par contradiction.

On ne peut pas coder le parcours en largeur sans utiliser explicitement une file.

Une erreur très commune consiste à penser qu'avec la récursivité, on va pouvoir coder un parcours en largeur. C'est peine perdue : la récursivité s'appuie sur une pile et non pas une file. Toute tentative de parcourir un graphe récursivement résultera en un parcours en profondeur.

### Parcours en profondeur avec une pile

La stratégie du parcours en profondeur consiste à développer le premier noeud fils du noeud source, puis parmi les fils de ce premier noeud fils, à nouveau un fils etc. On s'éloigne ainsi de la source sur un premier chemin. Lorsqu'on tombe sur une impasse, on remonte au dernier point de choix, et on choisit le fils suivant qui n'a pas été développé.

On va ainsi balayer un arbre des choix de gauche à droite comme sur la figure ci-dessous :

La différence avec la stratégie du parcours en largeur est très simple : lorsqu'on développe le premier noeud fils, les fils de rang deux sont examinés de suite au lieu d'être examinés après les frères des files de

rang 1. L'implémentation s'en suit : il suffit de ranger les noeuds à explorer dans une pile plutôt que dans une file pour que le premier soit choisi immédiatement.

Il est remarquable d'observer qu'en modifiant très simplement l'algorithme du parcours en largeur et en remplaçant la file par une pile, on obtient un autre algorithme de parcours avec des propriétés très différentes.

La structure identique au parcours en largeur implique que la complexité est identique, en  $O(|V| + |E|)$ .

## Parcours en profondeur récursif

Bien qu'on puisse coder le parcours en profondeur comme au paragraphe précédent en s'appuyant explicitement sur une pile, un codage plus naturel et avec quelques autres propriétés repose sur l'utilisation de la récursivité. Ici on scinde le parcours de graphe en deux parties : une fonction récursive qui visite une partie connexe du graphe à partir d'un noeud non exploré, et une partie qui se charge de recommencer cette exploration pour chaque noeud non encore exploré.

Cette procédure DFS-Visit() va retourner une forêt : si le graphe n'est pas connexe (cas d'un graphe non-orienté) ou si certaines parties ne sont pas accessibles depuis le noeud source (cas d'un graphe orienté), ces parties seront tout de même explorées, puisqu'on reprendra l'exploration à partir d'un noeud non encore visité, jusqu'à ce qu'il n'en reste aucun.

Le graphe des prédécesseurs ( $V_P, E_P$ ) construit par ce parcours est défini par :

$$V_P = V, E_P = \{(v.parent, v) | v \in V \wedge v.parent \neq \text{nil}\}$$

Cette procédure de parcours utilise un compteur global de «temps» qui enregistre l'instant auquel on entre dans un noeud et surtout l'instant auquel on en sort.

## Tri topologique

Une application classique du parcours en profondeur consiste à effectuer un tri topologique d'un graphe orienté acyclique ou DAG (*Direct Acyclic Graph*).

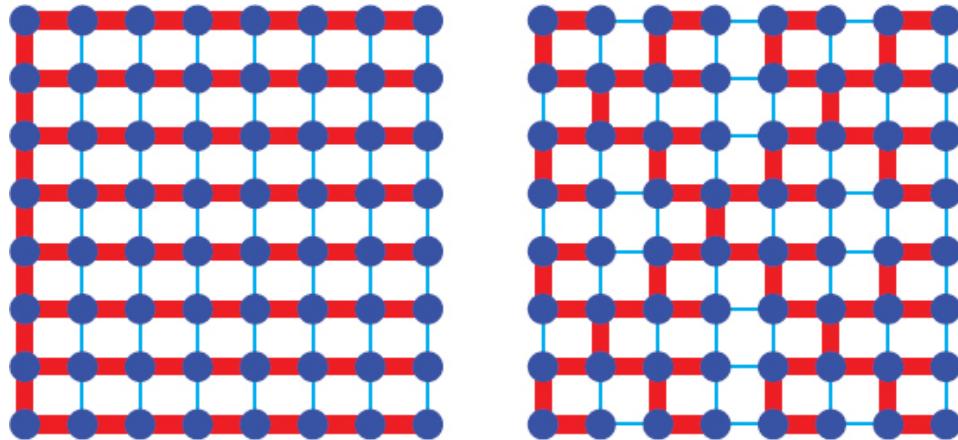
Ce tri topologique consiste à trouver trouver un ordonnancement linéaire des noeuds du graphe tel que pour tout  $(u, v) \in E$ ,  $u$  apparaît avant  $v$  dans cet ordonnancement. C'est une tâche courante pour un ordinateur qui doit trouver dans quel ordre charger fichiers en fonction d'une de relation de précédence par exemple. Bien évidemment, si le graphe comporte des cycles, un tel ordonnancement n'existe pas.

Ce problème est facilement résolu par un parcours en profondeur du graphe.

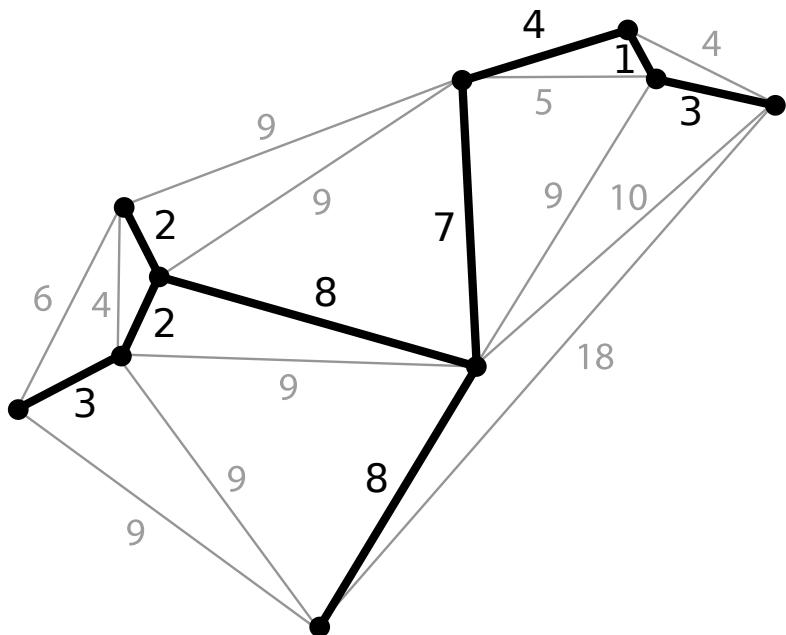
```
function TopologicalSort(G)
    #c Calculer les valeurs v.finish
    #c pour chaque noeud de V
    DFS(G)
    #c Insérer chaque noeud v au début d'une liste
    #c à l'instant où on ferme ce noeud
    return liste des noeuds
end
```

## 19.2 Arbres recouvrants de poids minimal

### Motivation



Sur un circuit imprimé, des composants électroniques doivent être interconnectés. Pour connecter électriquement un ensemble de  $n$  broches (*pins* en anglais) de composants, il faut choisir un arrangement de  $n - 1$  connections, chaque connection connectant 2 broches. Parmi l'ensemble des arrangements, celui utilisant la plus faible longueur de piste est préférable.



Pour interconnecter des bureaux à l'aide d'un réseau informatique, il faut choisir un sous-ensemble du graphe (potentiellement : du graphe complet) des connections de telle sorte que tous les bureaux soient connectés, mais aussi que la longueur de câblage soit minimale.

Les arbres recouvrants de poids minimal ont d'autres applications en traitement de signal (entropie de Renyi pour les images), en routage de données (protocoles d'autoconfiguration pour Ethernet), etc.

## Définition

Soit  $G = (V, E)$  un graphe et  $w$  une fonction de poids pour les arêtes de ce graphe.

On appelle arbre recouvrant de poids minimal un sous-ensemble acyclique  $T \subseteq E$  qui connecte tous les noeuds et dont le poids  $w(T) = \sum_{(u,v) \in T} w(u,v)$  soit minimal.

Puisque  $T$  est acyclique et connecte tous les noeuds, il doit former un arbre, que nous appelons un arbre *recouvrant*, car il recouvre tous les noeuds du graphe. De plus cet arbre est de poids minimal parmi tous les arbres recouvrant le graphe.

## Algorithme de Prim

En pseudo-code :

```
function MSTPrim(G, w, r)
    for u ∈ V(G)
        u.key ← ∞
        u.parent ← NIL

    r.key ← 0
    #c File de priorité
    Q ← V(G)
    while Q ≠ ∅
        u ← ExtractMin(Q)
        for v ∈ Adj(u)
            if v ∈ Q ∧ w(u,v) < v.key
                v.parent ← u
                v.key ← w(u, v)
    end
```

L'algorithme de Prim pour déterminer un arbre recouvrant de poids minimal utilise un noeud de départ  $s$  qui peut être choisi arbitrairement. Cet algorithme maintient une partition de l'ensemble  $V$  des noeuds du graphe en deux sous-ensembles :  $S$  et  $V \setminus S$ . Initialement, le noeud de départ est placé dans  $S$  :  $S = \{s\}$ .

La partition définit une *coupure* du graphe : l'ensemble des arêtes qui relient un noeud de  $S$  à un noeud de  $V \setminus S$ . Parmi l'ensemble de ces arêtes, on choisit celle de poids le plus faible et on l'ajoute à l'arbre recouvrant de poids minimal  $A$  en construction. Cette arête relie  $s$  à un noeud  $u \in V \setminus S$ . On place  $u$  dans  $S$ . La coupure se déplace et on itère le processus jusqu'à ce que  $S = V$ . À ce point, l'arbre  $A$  connecte tous les noeuds du graphe.

Une proposition de codage en Python :

```
from heapq import heappop, heappush

def prim(G, s):
    P, Q = {}, [(0, None, s)]
    while Q:
        _, p, u = heappop(Q)
        if u in P: continue
        P[u] = p
        for v, w in G[u].items():
            heappush(Q, (w, u, v))
    return P
```

## Algorithme de Kruskal

En pseudo-code :

```

function MSTKruskal(G, w)
    A ← Ø
    for v ∈ V(G)
        MakeSet(v)
    #c classer les arêtes en ordre
    #c de poids croissant
    E ← sort(E(G))
    for (u,v) ∈ E
        if FindSet(u) ≠ FindSet(v)
            A ← A ∪ {(u,v)}
            Union(u, v)
    return A
end

```

L'algorithme de Kruskal fonctionne en partant des  $|V|$  singletons (sous-ensembles disjoints) correspondant à chaque noeud. On va progressivement connecter des sous-ensembles de noeuds disjoints jusqu'à ce que tous les noeuds soient connectés. À chaque fois, on va chercher à connecter les deux sous-ensembles disjoints qui sont connectables par l'arête de plus faible poids. En pratique, on examine les arêtes par ordre de poids croissant. Chaque fois qu'une arête connecte deux sous-ensembles disjoints, on réunit ces deux sous-ensembles.

Une implémentation naïve en Python :

```

# coding: utf-8

# C est un dictionnaire de classes de noeuds.
# Tous les noeuds ont un représentant, initialement eux-mêmes.

# Trouver un représentant de la classe
def naive_find(C, u):
    while C[u] != u:
        u = C[u]
    return u

def naive_union(C, u, v):
    # Trouver les deux représentants
    u = naive_find(C, u)
    v = naive_find(C, v)
    # Les réunir : l'un réfère à l'autre
    C[u] = v

# Le graphe est un dictionnaire
# Chaque G[u] est aussi un dictionnaire
# tel que G[u][v] donne le poids de l'arête (u,v)
def naive_kruskal(G):
    # Ensemble des arêtes du graphe

```

```

E = [(G[u][v], u, v) for u in G for v in G[u]]
# Solution partielle, initialement vide
T = set()
# Classes des noeuds connectés
C = {u:u for u in G}
# Arêts, classées par poids
for _, u, v in sorted(E):
    if naive_find(C, u) != naive_find(C, v):
        # Deux noeuds avec des représentants différents
        # On retient l'arête
        T.add((u, v))
        # Il faut les connecter
        naive_union(C, u, v) # Combine components
return T

```

Critique : dans cette implémentation, l'opération `find()` peut nous ramener à parcourir une chaîne linéaire de références pour trouver le représentant de la classe. Le problème provient de la réalisation de l'opération `union()` qui ne différencie pas la taille des ensembles à réunir. Une idée triviale consiste à dire que lors de l'union, c'est le plus petit des deux ensembles qui doit référer au plus grand, de façon à *balancer* les longueurs des chaînes de référence.

Il y a plusieurs façons de réaliser cette amélioration. L'une d'entre elles passe par l'utilisation d'arbres binaires pour représenter les ensembles mis-en jeu par les opérations union-find. L'autre repose sur une technique astucieuse de compression des chemins et donne lieu au code Python suivant.

```

# coding: utf-8

# C est un dictionnaire de classes de noeuds.
# Tous les noeuds ont un représentant, initialement eux-mêmes.
def find(C, u):
    if C[u] != u:
        # Compression du chemin
        C[u] = find(C, C[u])
    return C[u]

# Ici C est un dictionnaire des classes des noeuds
# et R indique le rang d'un noeud
def union(C, R, u, v):
    u, v = find(C, u), find(C, v)
    # Union par le rang du noeud
    if R[u] > R[v]:
        C[v] = u
    else:
        C[u] = v
    # Même rang, il faut monter v d'un niveau
    if R[u] == R[v]:
        R[v] += 1

def kruskal(G):

```

```

E = [(G[u][v], u, v) for u in G for v in G[u]]
T = set()
# Les composants initiaux, avec leur rang
C, R = {u:u for u in G}, {u:0 for u in G}
for _, u, v in sorted(E):
    if find(C, u) != find(C, v):
        T.add((u, v))
        union(C, R, u, v)
return T

```

### 19.3 Recherche des plus courts chemins

Soit un graphe  $G = (V, E)$  orienté muni d'une fonction de poids  $w : E \rightarrow \mathbb{R}$ .

Le poids d'un chemin  $p = (v_0, \dots, v_k)$  est la somme des poids des arêtes qui le constituent :

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

On définit la distance de plus court chemin entre  $u$  et  $v$  par :

$$\delta(u, v) = \begin{cases} \min\{w(p) \mid u \rightsquigarrow v\} & \text{si il existe un chemin de } u \text{ à } v \\ \infty & \text{sinon.} \end{cases}$$

Un **plus court chemin** entre  $u$  et  $v$  est alors défini comme tout chemin  $p$  tel que  $w(p) = \delta(u, v)$ .

La recherche du plus court chemin est un problème d'**optimisation**.

Le problème de la recherche d'un plus court chemin entre les noeuds d'un graphe peut se diviser en trois problèmes :

- plus court chemin entre une source et une destination
- plus courts chemins entre une source et toutes les destinations
- plus courts chemins entre toutes les sources et toutes les destinations

#### Propriétés des plus courts chemins

Soit  $p = (v_0, \dots, v_k)$  un plus court chemin entre les noeuds  $v_0$  et  $v_k$  et  $p_{ij} = (v_i, v_{i+1}, \dots, v_j)$  pour tout  $i$  et  $j$  tels que  $0 \leq i \leq j \leq k$ . Alors  $p_{ij}$  est un plus court chemin de  $v_i$  à  $v_j$ .

La preuve s'obtient par contradiction en décomposant le chemin  $p$  en  $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$ . Le poids  $w(p)$  de  $p$  vaut  $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$ . Supposons qu'il existe un chemin  $p'_{ij}$  entre  $i$  et  $j$  avec un poids  $w(p'_{ij}) < w(p_{ij})$ . Alors le chemin  $p'$  défini par  $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$  serait un plus court chemin entre  $v_0$  et  $v_k$  et son poids serait  $w(p') = w(p_{0i}) + w(p'_{ij}) + w(p_{jk}) < w(p)$ , ce qui contredit l'hypothèse.

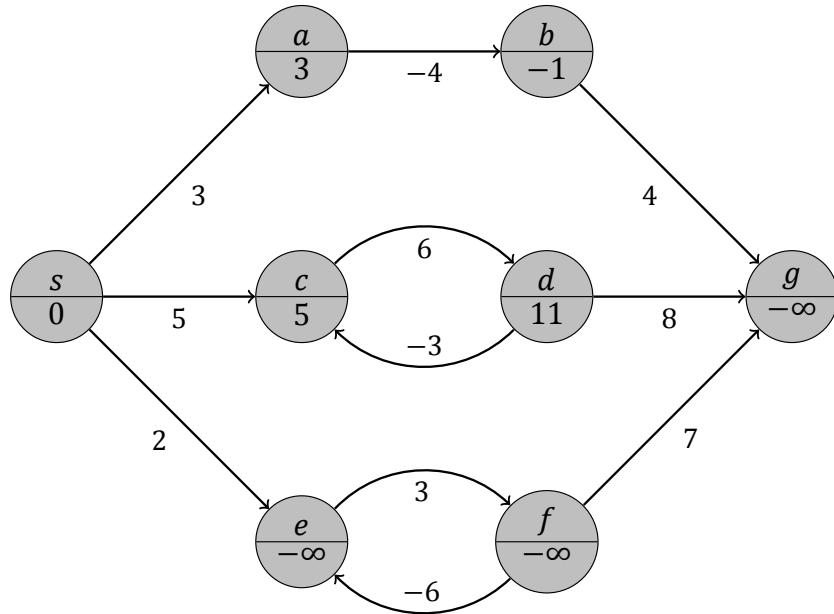


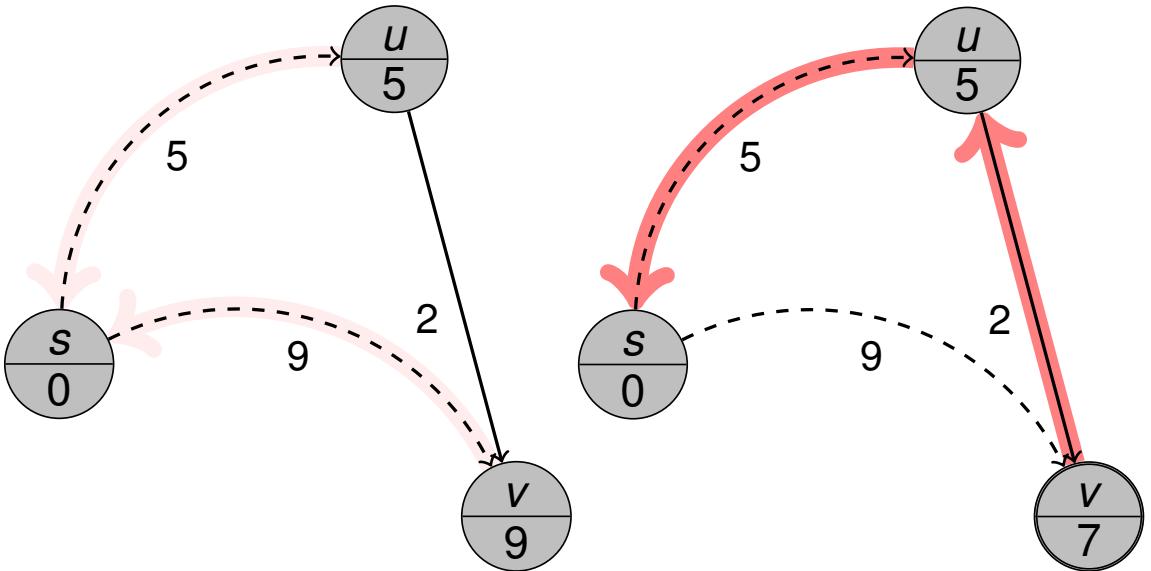
FIGURE 19.1 – plus court chemin et cycles

En toute généralité, un graphe peut contenir des arêtes de poids négatif. Les arêtes de poids négatif ne posent pas de problème tant qu'il n'y a pas de cycle de poids négatif atteignable depuis la source. Si il existe un cycle de poids négatif atteignable depuis la source, le chemin le plus court n'est plus défini : on peut toujours trouver un chemin plus court en « tournant » une fois de plus dans le cycle. Si il existe un cycle de poids négatif entre la source  $s$  et un certain noeud  $v$ , on pose  $\delta(s, v) = -\infty$ .

Un plus court chemin entre deux noeuds peut-il contenir un cycle ?

- Un plus court chemin ne peut pas contenir de cycle de poids strictement positif : on pourrait enlever le cycle pour obtenir un chemin de poids strictement inférieur.
- Un plus court chemin ne peut pas contenir de cycle de poids strictement négatif : le plus court chemin n'est pas défini.
- Il reste les cycles de poids 0 :
  - On peut enlever les cycles de poids 0 de n'importe quel chemin pour produire un chemin de même poids
  - Donc, si il existe un plus court chemin entre la source  $s$  et un noeud  $v$  qui contient un cycle de poids 0, alors il existe un autre chemin de même poids sans ce cycle.
  - On peut itérativement supprimer les cycles de poids 0 jusqu'à obtenir un chemin sans cycle.
- On peut donc se contenter de chercher les chemins les plus courts sans cycle, sans perte de généralité.
- Puisque tout chemin sans cycle dans un graphe  $G = (V, E)$  contient au plus  $|V|$  noeuds distincts, il contient également au plus  $|V| - 1$  arêtes.
- On peut donc se contenter d'examiner les chemins de longueur  $|V| - 1$  arêtes au plus !

Pour calculer les plus courts chemins, nous allons fonder nos algorithmes sur l'exploitation de la remarque suivante.



Sachant que la meilleure distance pour aller de  $s$  à  $u$  est 5 pour l'instant et que la meilleure distance pour aller de  $s$  à  $v$  pour l'instant est de 9, peut-on aller de  $s$  à  $v$  /en passant par  $u$  pour un coût meilleur? La réponse est oui dans ce cas : on peut réduire à 7 la distance de  $s$  à  $v$  en passant par  $u$ .

Nous allons utiliser les deux fonctions suivantes en pseudo-code :

```

function InitializeSingleSource(G, s)
    for v ∈ V(G)
        v.dist ← ∞
        v.parent ← nil
    s.dist ← 0
end

function Relax(u, v, w)
    #c w(u, v) est le poids de l'arête (u, v)
    if v.dist > u.dist + w(u, v)
        v.dist ← u.dist + w(u, v)
        v.parent ← u
    end
end

```

Nous allons faire l'hypothèse que le graphe est initialisé avec un appel à `InitializeSingleSource()` et par la suite, les seuls changements sont effectués par des appels à `Relax()`.

Par inégalité triangulaire, nous avons :  $\forall (u, v) \in E \quad \delta(s, v) \leq \delta(s, u) + w(u, v)$

Les valeurs des propriétés de distance (`dist` calculées) sont bornées par la distance de plus court chemin :  $\forall v \in V \quad v.dist \geq \delta(s, v)$  et lorsque  $v.dist$  atteint  $\delta(s, v)$ , cette valeur ne change plus.

Si il n'existe aucun chemin de  $s$  à  $v$ , alors  $v.dist = \delta(s, v) = \infty$ .

Le processus de relaxation converge : si  $s \rightsquigarrow u \rightarrow v$  est un plus court chemin dans  $G$  pour  $(u, v) \in V$ , et si  $u.dist = \delta(s, u)$  avant de relaxer l'arête  $(u, v)$ , alors  $v.dist = \delta(s, v)$  après la relaxation.

Relaxation de chemin : si  $p = (v_0, \dots, v_k)$  est un plus court chemin de  $v_0$  à  $v_k$  et que l'on relaxe les arêtes de  $p$  dans l'ordre  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  alors  $v_k.dist = \delta(s, v_k)$ . Ceci reste vrai même en présence d'autres opérations de relaxation d'autres arêtes entrelacées avec les opérations de relaxation du chemin  $p$ .

Sous-graphe des prédécesseurs : une fois qu'on a atteint  $v.dist = \delta(s, v)$  pour tous les  $v \in V$ , le graphe des prédécesseurs est un arbre des plus courts chemins enraciné en  $s$ .

### À partir d'une seule source : algorithme de Dijkstra

L'algorithme de Dijkstra repose sur une exploration du graphe à partir de la source dans un ordre particulier : les noeuds sont considérés par distance croissante au noeud source. Les noeuds sont conservés dans une file de priorité. Leur priorité est leur distance par rapport au noeud source. Ils sont extraits successivement dans cet ordre de la file de priorité. Les noeuds adjacents à chaque noeud  $u$  extrait sont relaxés : leur priorité est modifiée si on peut trouver un chemin plus court passant par  $u$ . Ceci implique que `Relax()` soit capable de modifier la priorité d'un noeud dans la file de priorité  $Q$ .

```

function Dijkstra(G, s, w)
    InitializeSingleSource(G, s)
    T  $\leftarrow \emptyset$ 
    #c Q est une file de priorité
    Q  $\leftarrow V(G)$ 
    while Q  $\neq \emptyset$ 
        u  $\leftarrow$  ExtractMin(Q)
        T  $\leftarrow T \cup \{u\}$ 
        #c pour chaque noeud
        #c v adjacent à u
        for v  $\in$  Adj[u]
            Relax(u, v, w)
    end

```

L'algorithme de Dijkstra donne les plus courts chemins entre le noeud  $s$  de départ et tous les autres noeuds. Ces chemins sont matérialisés par la relation parent des noeuds.

Propriété de l'algorithme : dès qu'un noeud sort de la file de priorité, son plus court chemin au noeud de départ est déterminé.

On peut prouver par induction l'invariant de boucle suivant : soit  $T$  l'ensemble des noeuds déjà visités,

- $\forall v \in T \quad v.dist = \delta(s, v)$
- $\forall v \notin T \quad v.dist$  est la longueur du plus petit chemin dans le sous-graphe  $T \cup \{v\}$ .

Enfin, l'algorithme suppose que  $w(u, v) \geq 0$ . En effet, pour que cette méthode de recherche de plus court chemin fonctionne, il faut que le poids le long d'un chemin soit une fonction croissante monotone. Si le poids pouvait décroître par ajout d'une valeur négative à un moment donné, le noeud atteint par cette arête de poids négatif pourrait avoir une distance à la source qui soit inférieure aux noeuds que l'on a déjà extraits de la file de priorité, ce qui devrait remettre en cause les plus courts chemins déjà calculés.

### Preuve de l'algorithme de Dijkstra

#### Complexité de l'algorithme de Dijkstra

L'algorithme de Dijkstra repose sur une file de priorité, et sa complexité dépend donc de la façon de réaliser cette file de priorité. C'est l'occasion de faire un curieux constat. La complexité d'un algorithme sur les graphes comme celui-ci dépend des deux paramètres  $|V|$  et  $|E|$ . Le choix optimal de la réalisation de la file de priorité dépend de la densité du graphe.

Il faut noter tout particulièrement que dans le cas de l'algorithme de Dijkstra, l'opération `Relax()` nécessite de modifier la distance du noeud  $v$  au noeud source, donc de modifier la priorité de ce noeud dans la file de priorité. Lorsque la file de priorité est réalisée par un tas, cette opération est réalisée par la fonction `HeapDecreasekey()`.

Si la file de priorité est réalisée par une simple liste :

- l'opération `ExtractMin(Q)` est réalisée en  $O(|V|)$ ,
- l'opération `Relax(u, v, w)` est réalisée en  $O(1)$ .

Il y a  $|V|$  opérations `ExtractMin(Q)` et  $|E|$  opérations `Relax(u, v, w)`. Le temps total d'exécution est donc en  $O(|V|^2 + |E|) = O(|V|^2)$ .

Si la file de priorité est réalisée par un tas minimal (`heapq` en Python) :

- l'opération `ExtractMin(Q)` est réalisée en  $O(\log |V|)$ ,
- l'opération `Relax(u, v, w)` est réalisée en  $O(\log |V|)$  car il faut restaurer la condition de tas minimal quand on change la valeur de la priorité d'un élément.

Il y a toujours  $|V|$  opérations `ExtractMin(Q)` et  $|E|$  opérations `Relax(u, v, w)`. Le temps total d'exécution est donc en  $O((|V| + |E|) \log |V|) = O(|E| \log |V|)$  (si tous les noeuds sont accessibles depuis la source).

Le tas minimal apporte donc un gain si et seulement si  $|E| < \frac{|V|^2}{\log |V|}$  : si le graphe est trop connecté, on appelle trop souvent `HeapDecreasekey(Q, v)` dans `Relax(u, v, w)` et le gain obtenu avec une opération `ExtractMin(Q)` en  $O(\log |V|)$  plutôt que  $O(|V|)$  s'efface.

Il est toutefois à noter que le cas des graphes très connectés apparaît peu fréquemment.

## Algorithme de Bellman-Ford

Cet algorithme calcule également les plus courts chemins entre un noeud source et les autres noeuds du graphe. Il résoud élégamment le problème de la détection des cycles de poids négatifs et des arêtes de poids négatif. Il relaxe tout simplement toutes les arêtes exactement  $|V| - 1$  fois, ce qui correspond à la plus grande longueur d'un plus court chemin entre deux noeuds du graphe. Ensuite, il essaie une nouvelle fois de relaxer l'ensemble des arêtes. Si aucune distance n'a bougé, c'est qu'on a bien déterminé les plus courts chemins entre le noeud source et les autres noeuds du graphe. Si certaines distances ont changé, c'est qu'il existe un cycle de poids négatif dans le graphe.

```
function BellmanFord(G, w, s)
    InitializeSingleSource(G, s)
    for i ← 1 to |V(G)| - 1
        for (u, v) ∈ |E(G)|
            Relax(u, v, w)
    for (u,v) ∈ |E(G)|
        if u.dist + w(u, v) < v.dist
            return false
    return true
```

Le coût d'exécution est plus élevé que celui de l'algorithme de Dijkstra. Il n'y a pas de file de priorité à gérer. Toutes les arêtes étant relaxées  $|V|$  fois, la complexité est en  $O(|V| \times |E|)$ .

## À partir de toutes les sources : algorithme de Floyd-Warshall

Si nous devons calculer les plus courts chemins entre toutes les paires de noeuds du graphe, devons-nous appliquer Dijkstra (ou Bellman-Ford)  $|V|$  fois, ou bien pouvons-nous faire mieux?

Soit  $V = (1 \dots n)$  les noeuds de  $G$  numérotés de 1 à  $n = |V|$ . Soit  $(1 \dots k)$  un sous-ensemble des noeuds de  $G$  pour  $k$  donné. Nous pouvons faire l'observation suivante.

Pour  $i, j \in V$ , considérons les chemins de  $i$  à  $j$  n'utilisant que les noeuds  $(1 \dots k)$ . Soit  $p$  un plus court chemin parmi ces chemins. Nous pouvons définir  $p$  en fonction des chemins entre  $i$  et  $j$  qui n'utilisent que  $(1 \dots k - 1)$ .

- si  $k$  n'est pas un noeud sur le chemin  $p$ , alors un plus court chemin entre  $i$  et  $j$  en utilisant seulement les noeuds de 1 à  $k$  est un plus court chemin entre  $i$  et  $j$  en utilisant seulement les noeuds de 1 à  $k - 1$ ,

- si  $k$  est sur le chemin  $p$ , nous décomposons  $p$  en  $i \xrightarrow{p_1} k \xrightarrow{p_2} j$ . Le chemin  $p_1$  est un plus court chemin entre  $i$  et  $k$  qui n'utilise que les noeuds  $(1 \dots k - 1)$  et  $p_2$  est un plus court chemin entre  $k$  et  $j$  qui n'utilise que les noeuds  $(1 \dots k - 1)$ .

Nous avons donc les équations de récurrence pour une approche par programmation dynamique !

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{si } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{si } k \geq 1 \end{cases}$$

Les pseudo-code de l'algorithme induit par ces équations est le suivant :

```
function FloydWarshall(W)
    n ← W.lignes
    D(0) = copie de W
    for k ← 1 to n
        D(k) ← nouvelle matrice n × n
        for i ← 1 to n
            for j ← 1 to n
                D(k)[i, j] ← min( D(k-1)[i, j],
                                    D(k-1)[i, k]
                                    +D(k-1)[k, j])
    return D(n)
end
```

Cet algorithme porte le nom de algorithme de Floyd-Warshall. Ces caractéristiques sont les suivantes :

- il supporte les arêtes de poids négatif
- il détecte les cycles de poids négatif
- il prend en entrée une matrice  $W$  telle que

$$w_{ij} = \begin{cases} 0 & \text{si } i = j \\ w(i, j) & \text{si } (i, j) \in E \\ \infty & \text{si } (i, j) \notin E \end{cases}$$

- il a une complexité en  $O(|V|^3)$
- attention : il est possible d'optimiser la place mémoire, mais il ne faut écraser des valeurs dont on aurait besoin ultérieurement.

## 19.4 Similitudes

Il est frappant d'observer la similitude de structure entre parcours en largeur, algorithme de Prim et algorithme de Dijkstra :

- les trois algorithmes ont strictement la même structure ;
- ils ne se différencient que par la nature de la structure de donnée auxiliaire : file ou file de priorité, et dans ce dernier cas, la clé utilisée :
  - BFS : file,
  - Prim : file de priorité avec les arêtes non-utilisées et leur poids comme clé,
  - Dijkstra : file de priorité avec les coûts des chemins développés comme clé ;
- BFS est identique à Dijkstra quand toutes les arêtes ont un coût unitaire.



**Cinquième partie**

**Problèmes**



## Problèmes sans solution

Beaucoup de problèmes naturels sont de nature binaire : la réponse est positive ou négative :

- est-ce qu'un nombre donné  $n$  est pair ?
- est-ce qu'un nombre donné  $n$  est premier ?
- est-ce qu'un graphe est planaire ?

On qualifie ce type de question de question binaire ou décisionnelle. Nombre de problèmes qui ne sont pas binaires peuvent se convertir simplement en questions décisionnelles. Par exemple, le fait de calculer  $\sqrt{n}$  n'est pas une question décisionnelle, mais on peut énumérer les entiers et tester pour chacun si son carré vaut  $n$ . La question décisionnelle  $k^2 = n$  peut donc servir de base à un algorithme de détermination de la racine carrée, et surtout à prouver son existence.

Les questions décisionnelles constituent donc une classe importante de questions auxquelles l'ordinateur devrait nous aider à apporter une réponse. Nous avons l'impression *a priori* qu'il doit exister un programme permettant de répondre à toute question décisionnelle. En fait nous allons voir qu'il n'en est rien. Ce problème est lié de très près au théorème d'incomplétude de Gödel et au fait que l'arithmétique n'est pas décidable.

Considérons tout d'abord l'ensemble des questions décisionnelles. Chaque question possède un ensemble de définition. Les données sont finies mais non bornées en taille. Chaque donnée peut être encodée en binaire sous forme d'un mot. Il s'agit donc de déterminer pour un mot s'il appartient à l'ensemble des instances positives ou négatives du problème. On éliminera les données qui ne correspondent pas à une instance du problème (il peut y en avoir). De façon symétrique, une partition d'un ensemble de mots définit un problème... même si on ne sait pas énoncer la question (cf. 42).

L'ensemble des mots sur un alphabet binaire est infini dénombrable : il s'agit des chaînes 0, 1, 00, 01, 10, 11, 000, 001, ... que l'on peut mettre en bijection avec  $\mathbb{N}$ . L'ensemble des problèmes décisionnels correspond à l'ensemble de toutes les partitions en deux sous-ensembles de cet ensemble de chaînes. Or, l'ensemble des parties d'un ensemble infini dénombrable est infini non-dénombrable : il a la puissance du continu, bien qu'étant infini, il est strictement plus grand qu'un ensemble infini dénombrable. (Voir la diagonale de Cantor).

L'ensemble des programmes informatiques est constitué de l'ensemble des chaînes finies non-bornées que l'on peut construire sur un alphabet fini (binaire par exemple). Cet ensemble de programmes informatiques est donc fini dénombrable.

Il y a donc strictement plus de problèmes décisionnels que de programmes informatiques. On peut donc soupçonner qu'il existe des problèmes décisionnels que l'ordinateur ne pourra pas résoudre. Ceci ne constitue pas en soi une preuve de ce fait, mais donne un indice sur l'origine de la difficulté : tout est lié à des problèmes de dénombrabilité.

On pourrait aussi se dire que les problèmes qui n'ont pas de solution algorithmique sont des objets étranges et peu intéressants dans la vie courante.

Nous allons montrer qu'un problème décisionnel simple et pratique ne peut être résolu par aucun algorithme. Cette démonstration est en fait fondée sur un théorème très intéressant : le théorème de récursion.

## 20.1 Le problème de l'arrêt

Imaginons que nous voulons détecter automatiquement des erreurs dans les programmes que les étudiants nous remettent à l'issue d'un contrôle. En particulier, nous cherchons à détecter que ces programmes s'arrêtent toujours, quelles que soient les données mises en entrée.

Par exemple, un étudiant pourrait nous envoyer le programme suivant :

```
def pourRire(x):
    if x == "lol": return pourRire(x)
    else: return "Ce n'était pas drôle"
```

Ce programme boucle si on le lance avec "lol" comme paramètre, sinon il s'arrête. Évidemment, un tel programme pourrait mettre en péril un système de correction automatique des devoirs, raison pour laquelle nous souhaitons développer un programme `verifArret` qui vérifie que tout programme s'arrête effectivement.

Notre programme `verifArret` devra prendre deux paramètres : un programme `p`, sous forme de chaîne de caractères et une chaîne de caractères `s` qui contiendra les données que nous souhaitons mettre en entrée de `p`. Ce programme `verifArret` renverra `True` si le programme `p` s'arrête pour les données contenues dans `s`, `False` sinon.

On pourrait imaginer qu'il suffise de simuler l'exécution de `p` sur les données de `s` ... mais ça ne peut pas fonctionner, puisque si `p` ne s'arrête pas, notre programme `verifArret` ne s'arrêtera pas non plus. Il faut donc que `verifArret` soit plus malin.

Examinons ce qui se passerait sur le programme `pourRire` :

```
>>> p = 'def pourRire(x):
...     if x == "lol": return pourRire(x)
...     else: return "Ce n'était pas drôle"'
>>> s = 'lol'
>>> verifArret(p, s)
False
>>> verifArret(p, 'bozo')
True
```

Supposons que `verifArret` existe. Nous allons nous en servir dans le programme suivant :

```
def paradox(p):
    if verifArret(p,p):
        while True:
            print "Je boucle !"
    else:
        return
```

Ensuite, plaçons ce programme lui-même dans une chaîne de caractères :

```
>>> p = 'def paradox(p):
...     if verifArret(p,p):
...         while True:
...             print "Je boucle !"
...     else:
...         return'
```

Et ensuite, utilisons cette chaîne comme argument de notre programme :

```
>>> paradox(p)
```

Que se passe-t-il ? Le programme `paradox` doit soit entrer dans une boucle infinie, soit s'arrêter. Ici, nous lui donnons comme paramètre une chaîne de caractères qui contient sa propre description. Il va donc appeler `verifArret` qui doit retourner `True` ou `False`.

- Si `verifArret` retourne `True`, cela signifie que le programme décrit par `p` va s'arrêter quand on lui donne `p` comme paramètre d'entrée. Autrement dit : le programme `paradox` s'arrête lorsqu'on lui donne sa propre description en entrée. Et c'est à ce moment que le programme `paradox` entre dans sa boucle infinie !
- Si `verifArret` retourne `False`, cela signifie que le programme `paradox` lorsqu'on lui donne sa propre description en entrée. Mais nous avons justement conçu le programme `paradox` pour qu'il s'arrête dans ce cas.

Nous avons donc une contradiction dans les deux cas. La seule hypothèse sur laquelle nous nous appuyons est l'existence du programme `verifArret`. Nous devons donc en conclure qu'il ne peut pas exister de programme `verifArret`.

## 20.2 Autres problèmes insolubles algorithmiquement

Il existe d'autres problèmes pratiques qui n'ont pas de solution algorithmique comme le [10ème problème de Hilbert](#).

### Le 10ème problème de Hilbert

Le 10ème problème de Hilbert concerne la recherche de solutions entières à des équations polynomiales à coefficients entiers. Hilbert demandait un processus algorithmique pour déterminer si un polynôme à nombre quelconque de variables et à coefficients entiers possédait ou non une solution en nombres entiers.

C'est un problème décisionnel : on doit fournir une réponse oui/non à chaque instance du problème. Youri Matiasssevitch a montré en 1970 qu'un tel algorithme était impossible.



# Problèmes faciles versus difficiles

## 21.1 Problèmes et instances

Un problème est défini par :

- des données en entrée, encodées selon un schéma manipulable par l'ordinateur (format informatique)
- une question sur les données en entrées.

Notre objectif est de concevoir un **algorithme** répondant (correctement) à la question.

Une **instance** d'un problème est définie en fixant un jeu de données en particulier.

Un **algorithme** produit (plus ou moins efficacement) une réponse à une instance du problème.

La complexité du problème est différente de la complexité d'un algorithme qui résoud le problème. La complexité du problème coïncide avec la complexité de son algorithme de résolution le plus efficace.

Un **problème de décision** suppose une réponse binaire oui / non. Un **problème d'optimisation** suppose la recherche d'une solution qui optimise une **fonction objectif**  $f : S \rightarrow \mathbb{N}$ . Un problème de décision *partitionne* l'ensemble des instances  $D$  en *instances positives*  $D^+$  et *instances négatives*  $D^-$ .

À chaque problème d'optimisation, nous pouvons associer un problème de décision dont la question est : « existe-t-il une solution pour laquelle la fonction objectif  $f$  est supérieure / inférieure à une valeur  $k$  donnée ? »

Exemples :

- plus court chemin : existe-t-il un chemin de longueur inférieure ou égale à  $k$  ?
- voyageur de commerce : existe-t-il un cycle hamiltonien de longueur inférieure ou égale à  $k$  ?
- rendu de monnaie : existe-t-il une solution nécessitant moins de  $k$  pièces ?

La complexité du problème d'optimisation est nécessairement supérieure ou égale à celle du problème de décision associé. En termes de classes de complexité : les deux problèmes sont comparables.

## 21.2 Classe P

La classe des problèmes  $\mathcal{P}$  est l'ensemble des problèmes de décision qui peuvent être résolus par un algorithme dont la complexité en temps est polynomiale en fonction de la taille des données :

$$\bigcup_{c>1} O(n^c)$$

La classe  $\mathcal{P}$  capture les problèmes « accessibles ». On n'a pas trouvé d'algorithme pratique avec des complexités supérieures à  $O(n^5)$ .

### 21.3 Classe NP

La classe des problèmes  $\mathcal{NP}$  est l'ensemble de problèmes de décision dont une solution (certificat) est vérifiable par un algorithme en temps polynomial.

- l'algorithme de vérification prend un certificat en entrée et répond oui / non au problème de décision
- l'algorithme de vérification est dans  $\mathcal{P}$
- pas de contrainte sur l'algorithme de recherche du certificat qui peut être exponentiel (ou plus!) : il doit trouver / construire le certificat.

Pour imaginer : il est facile de reconnaître un théorème en lisant sa preuve, mais il peut être difficile de trouver la preuve par nous même.

Exemples de problèmes NP :

- Existence de cycle hamiltonien dans un graphe : la vérification du certificat (séquence de noeuds) se fait en temps linéaire
- SAT : l'évaluation d'une formule propositionnelle en temps linéaire
- Sac à dos
- ...
- <http://www.mathpuzzle.com/eternity.html>
- ...

On ne sait pas si :

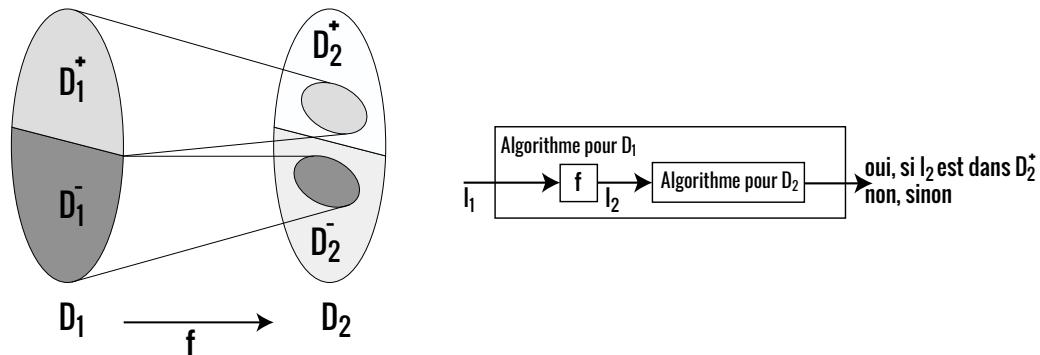
$$\mathcal{P} \stackrel{?}{=} \mathcal{NP}$$

C'est une des questions théoriques dont la réponse est la plus attendue. Personne n'croit que ces deux classes puissent être égales. En revanche, la preuve de l'inclusion stricte  $P \subset NP$  pourrait donner des indices théoriques importants sur la nature de beaucoup de problèmes.

### 21.4 Réduction polynomiale

Soit  $D_1$  et  $D_2$  deux problèmes de décision. On dit que  $D_1$  se réduit à  $D_2$  sous une réduction de Karp et on note  $D_1 \leq_K D_2$  si et seulement si il existe une fonction  $f$  telle que :

- $\forall I \in D_1 f(I) \in D_2$
- $I \in D_1^+ \Leftrightarrow f(I) \in D_2^+$
- $f$  est calculable en temps polynomial.



Autrement dit :  $D_1$  n'est pas plus difficile que  $D_2$  en termes de classe de complexité, i.e. à un facteur polynomial près.

## 21.5 NP-complétude

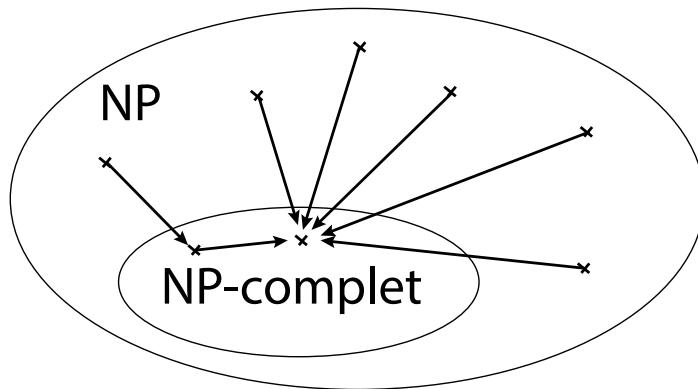
Un problème  $D$  est dit **NP-difficile** si  $\forall D' \in NP \quad D' \leq_K D$ .

Un problème  $D$  est dit **NP-complet** si  $D$  est NP-difficile et  $D \in NP$ .

Théorèmes :

- Si  $D \leq_K D'$  et  $D' \leq_K D''$  alors  $D \leq_K D''$ .
- Si  $D$  est NP-difficile et  $D \in P$  alors  $P = NP$ .
- Si  $D$  est NP-complet alors  $D \in P$  si et seulement si  $P = NP$ .

On peut réduire tout problème de NP à un problème NP-complet. Si un seul problème NP-complet est montré être dans P, alors la hiérarchie s'effondre.



Il existe des problèmes NP-complets. Le premier problème à avoir été prouvé NP-complet est SAT. Il s'agit du théorème de Cook-Levin (1971) : « Étant donné une formule propositionnelle à  $n$  variables propositionnelles, existe-t-il une assignation de ces variables aux valeurs de  $\{V, F\}$  qui rendent la formule logiquement équivalente à  $V$ ? »

Beaucoup d'autres problèmes ont été prouvés être NP-complets depuis.



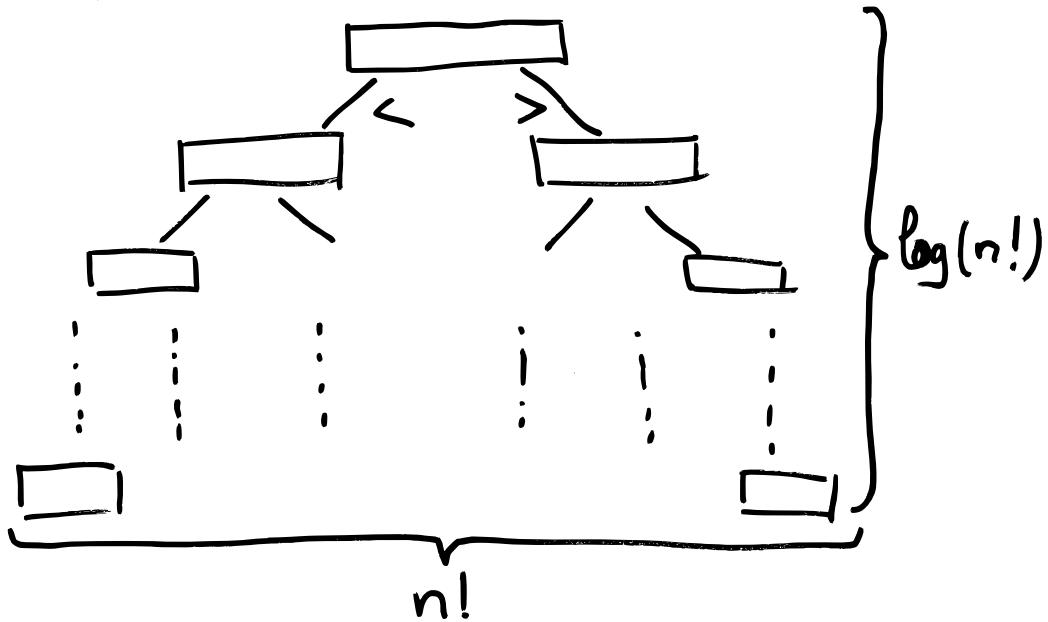
## Complexité de problème

Jusqu'ici, nous avons déterminé la complexité d'algorithmes particuliers résolvant un problème donné. Est-il possible de trouver une borne inférieure à l'ensemble des algorithmes qui résolvent un problème donné ? Si nous y arrivons, nous aurons déterminé la *complexité du problème* par rapport à la complexité des algorithmes qui le résolvent.

Un premier exemple trivial concerne le problème de déterminer le minimum d'un ensemble à  $n$  éléments non ordonnés. Nous supposons l'existence d'une relation d'ordre sur les éléments bien sûr. Tout algorithme visant à déterminer ce minimum devra effectuer  $n - 1$  comparaison au moins. La preuve s'obtient par l'absurde : si nous n'effectuons que  $n - 2$  comparaisons, nous avons omis de comparer deux éléments et nous ne pouvons donc pas savoir lequel est le plus petit. Notre algorithme ne résoud donc pas le problème. La complexité du problème de détermination du minimum est donc en  $O(n)$ .

Pour un exemple moins trivial, nous pouvons nous intéresser au cas du classement. Si nous supposons un classement qui repose sur une opération de comparaison binaire (cas habituel envisagé dans ce cours), nous pouvons prouver que la borne inférieure de la complexité de toute solution est en  $O(n \log n)$ .

En effet, l'objet du problème du classement consiste à déterminer quelle permutation des  $n$  éléments à classer respecte la relation ordre. Tout algorithme de classement devra décider sur la base d'une comparaison entre deux éléments vers quelle permutation se diriger. Il y a  $n!$  permutations possibles et le principe de décision binaire implique qu'on cherche dans un arbre binaire dont les feuilles sont ces permutations.



Au mieux, la hauteur de l'arbre binaire sera minimale pour supporter les  $n!$  feuilles et la hauteur de l'arbre donnera le nombre de comparaisons à effectuer pour trouver la bonne permutation. Si nous effectuons moins de comparaisons, nous allons ignorer certaines permutations et notre algorithme sera incorrect. La hauteur de l'arbre binaire est en logarithme de son nombre de feuilles, donc ici il faudra effectuer  $O(\log(n!))$  comparaisons. La formule de Stirling nous donne une approximation ( $O(\log(n!)) = O(n \log n)$ ). Tout algorithme de classement reposant sur la comparaison binaire de deux éléments s'exécutera donc au mieux en  $O(n \log n)$  comparaisons.

On a ici un exemple de détermination de la *complexité d'un problème* par opposition à la complexité d'un algorithme. En général, il est bien évidemment plus difficile de déterminer la complexité d'un problème que la complexité d'un algorithme particulier pour ce problème.

Par exemple, on ne connaît pas la borne inférieure de la complexité d'un algorithme pour multiplier deux matrices carrées de  $n \times n$ .

## **Sixième partie**

## **Annexes**



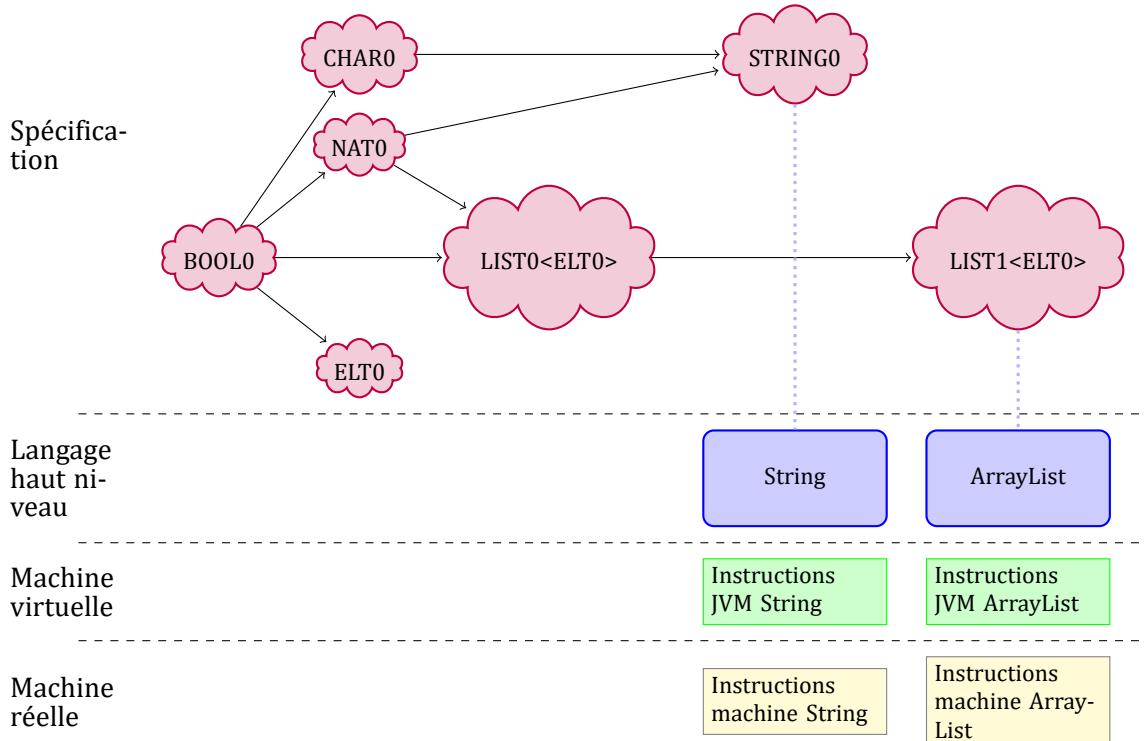
## Spécifications

Lorsqu'on construit du logiciel, il est intéressant de pouvoir prouver des propriétés sur le logiciel construit. En effet, on peut exiger qu'un programme auquel on fournit des données en entrées respecte certaines conditions sur les données fournies en sortie et essayer d'obtenir la preuve de la *correction formelle* du programme.

La théorie des *spécifications algébriques* utilise l'algèbre générale et la logique pour atteindre cet objectif.

Nous décomposerons la construction du logiciel selon deux dimensions :

- Structuration horizontale : nous allons classifier les objets manipulés par le programme et rassembler les objets identiques en *sortes*. Nous munirons ces sortes d'opérations. Nous allons ainsi modulariser la description de notre domaine. Les sortes sont hiérarchisées : on dira qu'une sorte  $s$  est plus générale qu'une sorte  $s'$  (et on note  $s' < s$ ) si  $s'$  supporte toutes les opérations de  $s$ . On dit alors que les opérations sont *polymorphes*. Nous allons ainsi obtenir des modules qui peuvent s'emboiter (un module sert à en définir un autre par extension du premier) ou se paramétriser (un module sert à compléter un autre module générique).
- Raffinement vertical : nous allons passer du niveau le plus abstrait où nous décrivons les propriétés de notre domaine au niveau concret des objets du langage de programmation par raffinements successifs. Les choix pour le raffinement seront guidés par des critères de rapidité, de place, de convivialité, etc.



Il existe plusieurs formalismes pour les spécifications formelles : certains universels (Z, VDM), d'autres plus spécifiques à certains domaines. Les spécifications algébriques définissent les objets par les opérations qui les engendrent ou les utilisent. Une théorie axiomatique complète la spécification en donnant les propriétés attendues des objets.

En pratique, il va falloir passer des objets abstraits de la spécification aux objets concrets du langage de programmation. De plus, il va falloir proposer des séquences d'instructions (fonctions) qui réalisent les spécifications sur les objets concrets. Une utilisation intelligente de la logique (logique de Hoare-Floyd) permet de prouver des propriétés sur un fragment de programme en pseudo-code.

La programmation doit se faire selon plusieurs critères : correction (par rapport aux specifications), robustesse (par rapport aux données en entrée), rapidité (exécution), économie (ressources), convivialité, maintenabilité, portabilité.

Il n'y a pas de langage de programmation meilleur qu'un autre en théorie : ils permettent tous de réaliser les mêmes calculs sur une machine donnée. Ce qui est possible avec l'un l'est aussi avec l'autre. Mais tous ne permettent pas de le faire aussi aisément ou avec la même fiabilité. Par exemple, les langages qui disposent d'une gestion automatique de la mémoire (garbage collection) causent moins d'erreur de programmation que ceux qui n'en ont pas. Il peut y avoir des avantages ou des inconvénients moins flagrants liés à certains types de langages.

Une **spécification algébrique** est définie par la donnée d'un ensemble de **sortes** d'éléments, de **profils d'opérations** et d'**axiômes**. La donnée des sortes et des profils d'opérations forme la **signature** de la spécification. Les axiômes imposent un comportement attendu des opérations.

Nous pouvons ainsi donner une spécification algébrique BOOL0 pour des objets booléens. Sur la base de cette spécification, on peut construire une nouvelle spécification NAT0 pour des entiers naturels.

**Spec :** BOOL0

**Sorts :** bool

**Operations :**

false :  $\rightarrow$  bool  
 true :  $\rightarrow$  bool  
 $_ \neg _$  : bool bool  $\rightarrow$  bool  
 $_ \wedge _$  : bool bool  $\rightarrow$  bool  
 $_ \vee _$  : bool bool  $\rightarrow$  bool

**Axioms :**

(b0) a, b, c : bool  
 (b1)  $\neg \text{true} = \text{false}$   
 (b2)  $\neg \neg a = a$   
 (b3)  $a \wedge \text{true} = a$   
 (b4)  $a \wedge \text{false} = \text{false}$   
 (b5)  $a \wedge (b \wedge c) = (a \wedge b) \wedge c$   
 (b6)  $a \wedge b = b \wedge a$   
 (b7)  $a \vee b = \neg((\neg a) \wedge (\neg b))$

**EndSpec :**

**Spec :** NAT0

**Extends :** BOOL0

**Sorts :** nat

**Operations :**

0 :  $\rightarrow$  nat  
 succ : nat  $\rightarrow$  nat  
 $_ + _$  : nat nat  $\rightarrow$  nat  
 $_ == _$  : nat nat  $\rightarrow$  bool  
 $_ < _$  : nat nat  $\rightarrow$  bool

**Axioms :**

(n0) m, n, p : nat  
 (n1)  $0 + n = n$   
 (n2)  $\text{succ}(m) + n = \text{succ}(m + n)$   
 (n3)  $(m + n) + p = m + (n + p)$   
 (n4)  $m + n = n + m$   
 (n5)  $0 == 0 = \text{true}$   
 (n6)  $\text{succ}(m) == 0 = \text{false}$   
 (n7)  $0 == \text{succ}(n) = \text{false}$   
 (n8)  $\text{succ}(m) == \text{succ}(n) = m == n$   
 (n9)  $0 < 0 = \text{false}$   
 (n10)  $\text{succ}(m) < 0 = \text{false}$   
 (n11)  $0 < \text{succ}(n) = \text{true}$   
 (n12)  $\text{succ}(m) < \text{succ}(n) = m < n$

**EndSpec :**

La spécification NAT0 étend la spécification BOOL0, car on a besoin des objets de sorte bool pour définir les objets de sorte nat.

On peut définir des spécifications *paramétriques* qui dépendent de la définition d'une autre spécification. Ici, par exemple, nous avons la spécification LIST0 d'objets de sorte list qui est paramétrée par la spécification ELT d'objets de sorte elt.

La spécification LIST0 construit des listes à l'aide de l'opération cons(). La liste [1, 2, 3] est construite comme cons(1, cons(2, cons(3, listevide))). Ceci est analogue à ce qui se passe pour les nombres de NAT0 ou les objets sont en fait succ(succ(0)) pour «2» ou bien encore succ(0)+succ(0) pour le même «2». Une problématique consiste à vérifier que 2 façons de construire un objet conduisent bien effectivement au même objet, i.e. que les axiômes permettent de prouver que succ(succ(0)) = succ(0)+succ(0).

Pour conclure, nous pouvons encore raffiner cette spécification des objets de sorte list en une spécification LIST2 qui formalise ce qu'est une *liste classée* par l'introduction du prédictat sorted().

Ici, nous avons fait l'hypothèse (Ass pour *assumes*) que la spécification des objets de sorte elt prévoyait l'existence d'une relation d'ordre entre éléments, ainsi que les axiomes correspondants (With).

L'utilisation du prédictat sorted() permettra éventuellement de prouver à l'aide de la logique de Hoare-Floyd qu'un algorithme de classement donné comme le classement par insertion, retournera toujours une liste classée.

**Attention :** ces techniques sont extrêmement lourdes et ne marchent bien que pour certains types de logiciels ou certaines parties de logiciel. Elles ne s'appliquent pas directement à un programme dans son intégralité. Malgré cette lourdeur, les techniques formelles ont permis de faire de grands progrès dans la fiabilité des logiciels complexes.

**Spec :** LIST0(ELT)  
**Extends :** NAT0

**Sorts :** list

**Operations :**

nil :  $\rightarrow$  list  
 cons( $\_$ ) : elt list  $\rightarrow$  list  
 length( $\_$ ) : list  $\rightarrow$  nat  
 head( $\_$ ) : list  $\rightarrow$  elt  
 tail( $\_$ ) : list  $\rightarrow$  list  
 append( $\_$ ,  $\_$ ) : list list  $\rightarrow$  list  
 rev( $\_$ ) : list  $\rightarrow$  list  
 revappend( $\_$ ,  $\_$ ) : list list  $\rightarrow$  list

**Axioms :**

xs, ys : list, x : elt

length(nil) = 0  
 length(cons(x, xs)) = 1 + length(xs)  
 head(cons(x, xs)) = x  
 tail(cons(x, xs)) = xs

append(nil, ys) = ys  
 append(cons(x, xs), ys) = cons(x, append(xs, ys))

**EndSpec :**

**Spec :** LIST2(ELT)  
**Extends :** LIST0(ELT)

**Assumes :**

( $<$ ) : elt, elt  $\rightarrow$  bool  
 ( $=<$ ) : elt, elt  $\rightarrow$  bool  
**With :** e,e1,e2,e3 :elt  
 (e  $<$  e) = false  
 (e1  $<$  e2)  $\Rightarrow$  (e2  $<$  e1) = false  
 (e1  $<$  e2)  $\wedge$  (e2  $<$  e3)  $\Rightarrow$  (e1  $<$  e3) = true  
 (e1  $=<$  e2) = (not (e2  $<$  e1))

**Operations :**

sorted : list  $\rightarrow$  bool

**Axioms :**

xs :list; x,y :elt  
 sorted(nil) = true  
 sorted(cons(x, nil)) = true  
 (x  $=<$  y)  $\Rightarrow$  sorted(cons(x, (cons y xs))) =  
 sorted(cons(y, xs))  
 (y  $<$  x)  $\Rightarrow$  sorted(cons(x, cons(y, xs))) = false

**EndSpec :**

## Compléments Python

### 24.1 Lambda expressions

Les lambda-expressions constituent un moyen simple d'utiliser une fonction *sans la nommer*. Lorsqu'on a besoin localement d'une fonction réalisant une opération élémentaire, il peut être pratique de ne pas définir explicitement une fonction connue globalement.

Ceci est extrêmement utile si on ajoute le fait que des variables peuvent référencer des fonctions et qu'on peut passer les fonctions en paramètre d'autres fonctions.

```
>>> def f (x): return x**2
...
>>> print f(8)
64
>>>
>>> g = lambda x: x**2
>>>
>>> print g(8)
64
```

On peut donner un exemple d'utilisation d'une fonction en argument. Soit une collection dont on ne veut garder que certains items vérifiant une propriété donnée. Par exemple, vous voulez filtrer les nombres pairs dans une collection de nombres, et les mots qui contiennent la lettre 'i' dans une collection de chaînes de caractères.

```
def filtre_nombres_pairs(seq, pred):
    ret = []
    for x in seq:
        if x % 2 == 0:
            ret.append(x)
    return ret
```

```
def filtre_mots_avec_i(seq, pred):
    ret = []
    for x in seq:
        if 'i' in x:
            ret.append(x)
```

```
    return ret
```

Le principe est identique dans les deux cas, mais le prédictat qui indique ce qu'il faut retenir est variable. On peut donc factoriser une partie du code et laisser variable le prédictat.

```
def filtre(seq, pred):
    ret = []
    for x in seq:
        if pred(x):
            ret.append(x)
    return ret
```

```
>>> filtre([1,2,3,4,5,6,7,8,9,10], lambda x: x % 2 == 0)
[2, 4, 6, 8, 10]
>>>
>>> filtre(['ananas', 'poire', 'pomme', 'kiwi'], lambda x: 'i' in x)
['poire', 'kiwi']
>>>
```

À noter que l'on peut encore réduire le code de la fonction filtre :

```
def filtre(seq, pred):
    return [x for x in seq if pred(x)]
```

On peut aller plus loin, parce que les fonctions définies localement par `lambda` peuvent *capturer* des variables de l'environnement :

```
def incrementeur(n):
    return lambda x: x + n
```

```
>>> g = incrementeur(3)
>>> h = incrementeur(4)
>>> g(3)
6
>>> h(3)
7
```

## 24.2 Exceptions

Le mécanisme des exceptions est un mécanisme parallèle à celui des appels de fonctions, mais destiné à récupérer un fonctionnement normal en cas d'erreur. Explication : le fonctionnement normal d'un programme est constitué par un enchaînement d'appels de fonctions. Chacune de ces fonctions est susceptible de déclencher une erreur si les paramètres qui lui sont fournis sont incorrects. Pour détecter ces erreurs, il faudrait donc que chaque fonction retourne une valeur conventionnelle indiquant "tout s'est bien passé", ou alors un code entier indiquant quelle erreur a été rencontrée. Le souci est que ces fonctions sont aussi supposées calculer un résultat qu'elles doivent renvoyer. On se retrouve donc devant le fait que chaque fonction doit renvoyer deux valeurs : un code d'erreur et un résultat, ce qui est encombrant à manipuler.

D'autre part, les appels de fonctions peuvent être imbriqués à un niveau assez profond. Le niveau auquel une erreur peut-être rattrapée est peut-être très supérieur au niveau auquel elle se produit. Il va donc falloir dans toutes les fonctions intermédiaires traiter le cas : est que ma valeur renournée comporte un code d'erreur et si oui le propager immédiatement à la fonction supérieure. On voit bien que ce mécanisme de traitement d'erreur polluerait rapidement n'importe quel programme.

Les exceptions fournissent un moyen élégant de contourner ce traitement des erreurs qui serait trop lourd. Il repose sur la notion d'état de la machine. À un instant donné, on mémorise l'état de la machine (registres et pile) et on indique que si une certaine erreur survient, on doit revenir immédiatement dans l'état mémorisé, en rompant le flot d'exécution normal du programme.

Exemple en Python :

```
def divise(x, y):
    return x / y

def intermediaire(x, y):
    print('début intermédiaire')
    z = divise(x, y)
    print('fin intermédiaire')
    return z

def calcule(x, y):
    z = intermediaire(x, y)
    return z
```

```
>>> print(calcule(3, 0))
début intermédiaire
Traceback (most recent call last):
  File "C:/temp/foo.py", line 16, in <module>
    print(calcule(3, 0))
  File "C:/temp/foo.py", line 13, in calcule
    z = intermediaire(x, y)
  File "C:/temp/foo.py", line 8, in intermédiaire
    z = divise(x, y)
  File "C:/temp/foo.py", line 4, in divise
    return x / y
ZeroDivisionError: division by zero
>>>
```

```
def divise(x, y):
    return x / y

def intermediaire(x, y):
    print('début intermédiaire')
    z = divise(x, y)
    print('fin intermédiaire')
    return z

def calcule(x, y):
```

```
try:  
    z = intermediaire(x, y)  
except ZeroDivisionError:  
    return None  
return z
```

```
>>> print(calcule(3, 0))  
début intermédiaire  
None  
>>>
```

Dans le deuxième cas, il n'y a pas d'erreur : la valeur `None` a été retournée. La fonction `intermediaire` a bien été court-circuitée : le traitement de l'exception a lieu dès que celle-ci se produit.

L'environnement fournit des exceptions prédéfinies pour des situations courantes : accès avec un index en dehors d'une liste, division par zéro, etc. On peut également définir ses propres exceptions déclencher ses propres exceptions. Se reporter à <https://docs.python.org/3.4/tutorial/errors.html> pour des exemples d'utilisation plus complexes.

# Programmation objet en Python

## 25.1 Introduction

Tout langage fournit des types de données, certains natifs, d'autres par l'intermédiaire de bibliothèques. La différence est d'ailleurs importante, même si elle ne saute pas aux yeux des novices. Un type de donnée natif sera toujours mieux intégré aux diverses constructions du langage.

Les langages informatiques sont extensibles, certains plus que d'autres. La plupart autorisent la définition de nouveaux types de données. Certains autorisent aussi la définition de nouvelles constructions du langage (ils sont beaucoup plus rares!).

La définition d'un nouveau type de données passe par deux étapes :

- l'allocation de blocs mémoire pour les "objets" de ce nouveau type que l'on veut manipuler
- la définition des opérations sur les objets en question.

Il existe un paradigme assez général pour implémenter de nouveaux types de données que l'on qualifie de *programmation orientée objet*. Ce paradigme possède plusieurs déclinaisons et il est traduit différemment selon les langages de programmation.

Nous allons donc donner ici une brève introduction à sa version Python qui sera suffisante pour nos besoins.

## 25.2 Classes et instances

À développer

## 25.3 Méthodes

À développer

## 25.4 Héritage

À développer



# Réalisation Python de structures de données classiques

L'intérêt est de réaliser ces structures de données en Python essentiellement pédagogique : ces structures sont fournies par des bibliothèques qui sont testées au niveau de leur correction et de leur performance.

D'autres langages que Python fournissent également ces structures de données.

## 26.1 Liste chaînée

On part de la spécification donnée.

Il faut créer deux classes : l'une pour les noeuds de la liste, l'autre pour la liste elle-même. On peut avoir envie de confondre la liste avec le noeud qui est en tête de liste et ne créer qu'une seule classe. Céder à cette tentation de facilité, c'est s'exposer par la suite à des problèmes lorsqu'on cherchera à étendre les fonctionnalités. Les classes doivent refléter la vraie nature de ce qu'elles modélisent. Une liste n'est pas un noeud.

La classe `LinkedList` doit refléter la signature de la spécification.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 26 17:38:15 2015

@author: Fabrice
"""

class Node:

    def __init__(self, v = None):
        self.prev = self.next = None
        self.value = v

    def __str__(self):
        ret = self.value.__str__()
        if self.next != None:
            ret = ret + ',' + self.next.__str__()
        return ret
```

```

class LinkedList:

    def __init__(self):
        self.head = None

    def __str__(self):
        return '[' + self.head.__str__() + ']'

    def insert(self, v):
        x = Node(v)
        x.next = self.head
        if self.head != None:
            self.head.prev = x
        self.head = x
        return self

    def delete(self, x):
        if x.prev != None:
            x.prev.next = x.next
        else:
            self.head = x.next
        if x.next != None:
            x.next.prev = x.prev
        return self

    def search(self, k):
        x = self.head
        while x != None and x.value != k:
            x = x.next
        return x

if __name__ == '__main__':
    L = LinkedList()
    L.insert(3).insert(5).insert(2).insert(4)
    print(L)
    x = L.search(2)
    print(x.value)
    L.delete(x)
    print(L)

```

## 26.2 Arbres binaires de recherche

À développer

## 26.3 Dictionnaires

Nous considérons que ce dictionnaire est une extension des tables à des clés quelconques. La spécification a été donnée auparavant et nous commençons donc par définir la classe `HashMap` dont la structure sera la suivante :

```
class HashMap:

    def __init__(self):
        pass

    def hash(key):
        pass

    def put(self, key, val):
        pass

    def get(self, key, val):
        pass
```

Pour définir une structure de hash-table, il faut se donner deux choses :

- une fonction de hash-age qui va associer un nombre à chaque clé,
- une taille pour la table.

Dans le cas le plus simple, une hash-table est de taille fixe. Si on la remplit de trop, les performances diminuent. L'extension automatique d'une hash-table est un problème assez difficile à résoudre de façon efficace. Nous allons donc nous cantonner ici à une hash-table de taille donnée.

```
class HashMap:

    def __init__(self, size):
        self.size = size
        self.table = [None * size]
```

Ensuite, il faut se donner la fonction de hash-age. Cette fonction doit associer un nombre à la suite d'octets qui compose la clé de l'objet.

```
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 22 13:05:48 2015

@author: Fabrice
"""

class HashTable:
    'Exemple de classe pour une table de hachage'
    tableSize      = 0
    entriesCount   = 0
    alphabetSize   = 256
```

```

hashTable      = []

def __init__(self, size):
    self.tableSize = size
    self.hashTable = [[] for i in range(size)]

def _hashing(self, key):
    hash = 0
    for c in key:
        hash += hash * self.alphabetSize + ord(c)
    return hash % self.tableSize

def put(self, key, value):
    hash = self._hashing(key)

    # Si un item avec la même clé existe,
    # il faut le remplacer
    for (i, (k, v)) in enumerate(self.hashTable[hash]):
        if k == key:
            del self.hashTable[hash][i]
            self.entriesCount -= 1
    self.hashTable[hash].append((key, value))
    self.entriesCount += 1
    return self

def get(self, key):
    hash = self._hashing(key)
    for (i, (k, v)) in enumerate(self.hashTable[hash]):
        if k == key:
            return self.hashTable[hash][i]
    print ("La clé " + key + " n'existe pas dans la table !")
    return None

def delete(self, key):
    hash = self._hashing(key)
    for (i, (k, v)) in enumerate(self.hashTable[hash]):
        if k == key:
            del self.hashTable[hash][i]
            self.entriesCount -= 1
    return self
    print ("La clé " + key + " n'existe pas dans la table !")
    return self

# C'est volontairement que ceci n'est pas une méthode
# __str__()
def print(self):

```

```
print (">>>> Contenu de la table >>>>")
print (str(self.getNumEntries()) + " entrées dans la table")
for i in range(self.tableSize):
    print (" [ " + str(i) + "] : ", end=' ')
    for j in range(len(self.hashTable[i])):
        print(self.hashTable[i][j], end=' ')
    print()
print ("<<<< Fin <<<<")
return self

def getNumEntries(self):
    return self.entriesCount

if __name__ == "__main__":
    hs = HashTable(11)

    hs.put("one", 1)
    hs.print()
    hs.put("one", 1)
    hs.print()

    hs.put("two", 2)

    hs.put("three", 3)
    hs.print()

    hs.put("one", 4);

    v = hs.get("one");
    print(v)

    hs.put("Uneclébeaucoupluslongue", 12345)
    hs.print()

    v = hs.get("Uneclébeaucoupluslongue")
    print(v)

    v = hs.get("Uneclélongue")
    print(v)

    hs.delete("Uneclébeaucoupluslongue")
    hs.print()

    hs.delete("Uneclélongue")
    hs.print()
```

```
hs.delete("un")
hs.print()
```

## 26.4 Tas

La structure de tas minimal `heapq` proposée par la bibliothèque Python est utilisable pour des files de priorité mais elle n'expose pas la méthode qui permettent de remettre le tas lorsqu'on a modifié la clé d'un élément : `HeapDecreaseKey`, or cette fonction nous a été nécessaire en particulier dans l'opération de relaxation pour trouver les plus courts chemins dans un graphe. En fait cette méthode existe (au moins à la date de la version 3.4 de Python) et porte le nom de `heapq._siftdown(heap, pos)`.

Vous trouverez ci-dessous une implémentation complète de la structure de tas qui suit la spécification donnée plus haut au chapitre sur les structures de données.

```
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 22 15:55:13 2015

@author: Fabrice
"""

import random

# Classe pour un tas
class minHeap:

    # Initialisation d'un tas, éventuellement
    # à partir d'une liste de valeurs
    def __init__(self, values=[]):
        self.tab = values
        self.buildMinHeap()
        pass

    # Indice du parent
    def parent(self, i):
        return i // 2

    # Indice du fils gauche
    def left(self, i):
        return i*2 + 1

    # Indice du fils droit
    def right(self, i):
        return i*2 + 2

    # Restauration de la condition de tas minimal
    # à l'indice i. On suppose que les sous-arbres
    # gauche et droit sont des tas minimaux.
```

```

def minHeapify(self, i):
    # Lequel entre i, l et r est l'élément minimal ?
    l = self.left(i)
    r = self.right(i)
    if l < self.size and self.tab[l] < self.tab[i]:
        smallest = l
    else:
        smallest = i
    if r < self.size and self.tab[r] < self.tab[smallest]:
        smallest = r
    # Si l'élément minimal n'est pas i
    if smallest != i:
        # On permute l'élément i avec l'élément minimal
        self.tab[i], self.tab[smallest] = self.tab[smallest], self.tab[i]
    ]
    # Et on n'oublie pas de restaurer la condition
    # de tas minimal dans le sous-arbre où on l'a détruite.
    self.minHeapify(smallest)

# Construction d'un tas minimal à partir
# d'éléments non ordonnés.
def buildMinHeap(self):
    self.size = len(self.tab)
    for i in range((self.size - 1)//2, -1, -1):
        self.minHeapify(i)
    pass

# Diminution de la valeur de la clé d'un élément
def heapDecreaseKey(self, i, v):
    if self.tab[i] < v:
        print('Attention : nouvelle valeur de la clé supérieure à l\'ancienne !')
        print('Requête ignorée.')
        return
    self.tab[i] = v
    while i >= 0 and self.tab[self.parent(i)] > self.tab[i]:
        self.tab[i], self.tab[self.parent(i)] = self.tab[self.parent(i)],
        self.tab[i]
        i = self.parent(i)

def push(self, value):
    self.tab.append(value)
    self.size += 1
    i = self.size - 1
    self.heapDecreaseKey(i, value)
    return self

```

