COMPLEXITE DES ALGORITHMES
METHODE DU SIMPLEXE

O. Marguin — 4/05/11

1. Généralités

(1.1) **Notion de complexité**

Un algorithme comporte :
- une partie *temporelle* : séquence d'instructions en principe savamment orchestrée,
- une partie *spatiale* : ensemble de données plus ou moins structurées (nombres, vecteurs, matrices, listes...).

L'algorithme a un *coût* qui est lié :
- au nombre d'opérations effectuées (opérations arithmétiques, logiques, transferts...),
- à l'espace-mémoire occupé par les données.

Evaluer ce coût revient à mesurer ce qu'on appelle la *complexité* de l'algorithme, en temps comme en espace. Il s'agit donc de dénombrer des opérations et des octets.

Lorsqu'on parle de *complexité* (tout court), on entend généralement complexité en temps. Il faut bien sûr préciser quelles sont les opérations à prendre en compte. Généralement, pour un algorithme numérique ce sont les opérations arithmétiques usuelles $(+, -, \times, \div)$; pour un algorithme de tri, ce sont les comparaisons entre éléments à trier (qui sont en principe les plus coûteuses). Les transferts de valeurs ou *affectations* sont souvent négligées. Dans toute la suite, nous écrirons ces affectations sous la forme $a \leftarrow b$, signifiant que la variable a reçoit la valeur de l'expression b.

(1.2) **Ordre de grandeur**

Soient deux fonctions f et $g : \mathbf{N} \to \mathbf{N}$. On dit que f et g ont *asymptotiquement le même ordre de grandeur*, et on note $f = \Theta(g)$, s'il existe deux constantes k, K strictement positives telles que, pour n assez grand, on ait :

$$k\, g(n) \leq f(n) \leq K\, g(n)$$

(ce qui équivaut à la double condition : $f = \mathrm{O}(g)$ et $g = \mathrm{O}(f)$).

Si n est une quantité liée à la taille des données (par exemple la dimension des vecteurs manipulés), on parle d'*algorithme de complexité* $g(n)$, ou *algorithme en* $g(n)$, lorsque sa complexité est de la forme $\Theta(g)$. Un algorithme en 1 (resp n) est dit *en temps constant* (resp. *linéaire*).

(1.3) **Optimalité**

Pour résoudre un problème donné, l'objectif est évidemment de trouver un algorithme *optimal*, c'est-à-dire dont la complexité soit d'ordre minimal. Comme exemple d'algorithme linéaire et optimal, on peut citer l'algorithme de Hörner permettant d'évaluer les polynômes de degré n. Le produit naïf des mêmes polynômes est en n^2. Le produit, le calcul de l'inverse, la factorisation LU... sur les matrices carrées d'ordre n sont en n^3, la résolution d'un système linéaire aussi. Il y a des problèmes difficiles qu'on ne sait pas résoudre en temps polynômial (par exemple le fameux problème du *voyageur de commerce*).

Améliorer l'efficacité d'un algorithme, c'est-à-dire en diminuer la complexité, est souvent très délicat. Par exemple, au prix de certaines astuces (voir [3], § 2.11), *Strassen* a trouvé un algorithme permettant de multiplier deux matrices $n \times n$ (de même que d'inverser une telle matrice) en $n^{\log_2 7} \simeq n^{2,8}$, ce qui est un peu mieux que n^3. Nous verrons plus tard qu'on peut multiplier deux polynômes de degré n en $n \log n$ grâce à l'algorithme de *transformation de Fourier rapide* (*FFT*).

(1.4) La complexité d'un algorithme, dans le cas où les données sont les plus favorables à son accomplissement, s'appelle **complexité dans le meilleur des cas**. On définit de même la **complexité dans le pire des cas** ainsi que la **complexité en moyenne**, où on considère que toutes les données possibles sont équiprobables. L'évaluation de ces trois types de complexité est importante pour juger par exemple de l'efficacité d'un algorithme de tri : voir § 4.

(1.5) **Stratégie "diviser pour conquérir"**

On a un problème à résoudre sur une structure de taille n. L'idée consiste à ramener sa résolution à :
- la résolution du même problème sur plusieurs structures de taille divisant n,
- puis au "recollement" de ces solutions pour obtenir la solution cherchée.

En notant $C(n)$ la complexité pour la taille n, on a alors une récurrence du type :

$$C(n) = a\,C(n/b) + f(n)$$

où a et b sont des entiers, $a \geq 1, b > 1$ et où on interprète n/b soit par $\lfloor n/b \rfloor$, soit par $\lceil n/b \rceil$.

Théorème :

Avec ces notations, si $f(n) = \Theta(n^{\log_b a})$, *alors* $C(n) = \Theta(n^{\log_b a} \log n)$ *; si* $f(n) = \mathrm{O}(n^{\log_b a - \varepsilon})$ *avec* ε *constante* > 0, *alors* $C(n) = \Theta(n^{\log_b a})$.

(pour une preuve: voir [1], théorème 4.1).

Dans le premier cas avec $a = b = 2$, on obtient $C(n) = \Theta(n \log n)$. Nous verrons un exemple d'application de cette stratégie au § 5.

2. Recherche dans un tableau

Il s'agit d'écrire un algorithme permettant de détecter la présence d'un élément x dans un tableau $T[1..n]$ de taille n.

(2.1) Considérons l'algorithme (dit *du drapeau*) :

```
existe ← faux
pour i de 1 à n faire
        si x = T[i] alors                (*)
                existe ← vrai
        fin du si
fin du pour
```

et prenons en compte uniquement les comparaisons effectuées à la ligne (*). Les complexités dans le pire des cas, dans le meilleur des cas et en moyenne sont toutes trois égales à n.

(2.2) Pour le même problème, l'algorithme suivant, qui comporte une boucle à deux sorties, est plus efficace :

```
existe ← faux
pour i de 1 à n tant que non existe faire
        existe ← x = T[i]
fin du pour
```

car il est facile de voir que sa complexité est :
- dans le pire des cas : n,
- dans le meilleur des cas : 1,
- en moyenne : $\frac{(n+1)}{2}$.

3. Opérations sur les entiers

Dans ce paragraphe, nous envisageons les opérations arithmétiques sur les entiers naturels, telles que nous avons l'habitude de les effectuer "à la main", c'est-à-dire chiffre par chiffre. On suppose que les nombres sont écrits en base 10 (chiffres de 0 à 9).

(3.1) **Addition, multiplication, division**

Considérant comme opérations élémentaires les opérations chiffre à chiffre, il est clair que l'addition de deux nombres de longueur n est de complexité $\Theta(n)$. La multiplication d'un nombre de longueur n par un chiffre est également de complexité $\Theta(n)$. On en déduit facilement que la multiplication de deux nombres de longueur n est de complexité $\Theta(n^2)$. La division euclidienne, telle qu'on l'effectue manuellement, est aussi de complexité $\Theta(n^2)$.

(3.2) **Dualité temps ↔ espace**

Il est frappant de constater que, vis-à-vis de la complexité, les parties temporelles et spatiales d'un algorithme se comportent à la manière de vases communiquants : si l'on gagne en complexité sur l'une, on perd sur l'autre et inversement. Par exemple, les algorithmes de multiplication et division mentionnés ci-dessus sont peu coûteux en espace (quelques tableaux de chiffres suffisent). Si l'on en développe la partie spatiale, on peut en diminuer la complexité en temps. Ainsi, on peut ramener la multiplication d'un nombre de longueur quelconque par un chiffre à une opération en temps nul ! C'est le procédé astucieux inventé par *Genaille* au $19^{\text{ème}}$ siècle, qui donne le résultat sans aucun calcul, par simple juxtaposition de réglettes. Il en est de même de la division d'un nombre de longueur quelconque par un chiffre, obtenue instantanément grâce aux *réglettes trisectrices de Genaille et Lucas*. Nous donnons ces étonnantes réglettes en annexes 1 et 2.

Il faut voir ces réglettes comme des tables de multiplication et de division "généralisées" d'un nombre quelconque par un chiffre. Bien sûr, la programmation de l'algorithme sous cette forme spatialisée nécessite une structure de données complexe (un *graphe*), impliquant des structures chaînées : si au final la complexité en temps est nulle pour ce qui est des opérations arithmétiques, en réalité le parcours d'un chemin dans ce graphe ne se fait pas en temps zéro.

4. Tri élémentaire

On a une relation d'ordre définie sur certains objets, et un tableau contenant de tels objets. Un *algorithme de tri* est un algorithme permettant de ranger les éléments de T par ordre croissant (ou décroissant).

(4.1) **Tri par insertion**

Soit $T[1..n]$ un tableau contenant n valeurs. Le *tri par insertion* de T se fait par l'algorithme :

pour i variant de 2 à n faire { insérer $T[i]$ dans $T[1..i-1]$ supposé déjà trié, c'est-à-dire : }
  $aux \leftarrow T[i]$
  $j \leftarrow i-1$
  tant que $(j > 0)$ et $(T[j] > aux)$ faire $(*)$
    $T[j+1] \leftarrow T[j]$
    $j \leftarrow j-1$
  fin du tant que
  $T[j+1] \leftarrow aux$
fin du pour

(4.2) **Etude de complexité**

On se propose de calculer le nombre c_n de comparaisons entre éléments du tableau effectuées à la ligne $(*)$ (c'est-à-dire le nombre de tests $T[j] > aux$). Sans restreindre la généralité, on peut supposer qu'au départ T contient les n entiers $1, 2, \ldots, n$ dans un ordre quelconque.

a) *complexité dans le pire des cas* (T initialement rangé par ordre décroissant) :

$$c_n = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

c'est-à-dire $\Theta(n^2)$.

b) *complexité dans le meilleur des cas* (T initialement rangé par ordre croissant) :

$$c_n = 1 + 1 + \cdots + 1 \ (n-1 \text{ fois}) = n - 1.$$

c'est-à-dire $\Theta(n)$.

c) *complexité en moyenne* : soit σ la permutation de $\{1, \ldots, n\}$ représentée par T. Comme il y a $n-1$ comparaisons forcées, on a :

$$c_n = (n-1) + \text{inv}(\sigma) \tag{1}$$

où $\text{inv}(\sigma)$ est le nombre d'inversions de σ, i.e. le nombre de couples d'entiers (i, j) tels que $1 \leq j < i \leq n$ et $\sigma(j) > \sigma(i)$. Tout revient donc à calculer le nombre moyen d'inversions d'une permutation arbitraire σ de $\{1, \ldots, n\}$. Pour cela, pour $1 \leq j < i \leq n$, soit X_{ij} la variable aléatoire définie par :

$$\begin{cases} X_{ij} = 1 & \text{si } \sigma(j) > \sigma(i) \\ \quad = 0 & \text{sinon.} \end{cases}$$

Par définition, l'espérance de X_{ij} est :

$$E(X_{ij}) = 1 \times \frac{1}{2} + 0 \times \frac{1}{2} = \frac{1}{2}$$

car toutes les permutations sont supposées équiprobables, et :

$$\text{inv}(\sigma) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} X_{ij}.$$

En passant aux espérances :

$$E(\text{inv}(\sigma)) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} E(X_{ij}) = \mathrm{C}_n^2 \times \frac{1}{2} = \frac{n(n-1)}{4}.$$

D'après (1), la complexité en moyenne est finalement :

$$c_n = (n-1) + \frac{n(n-1)}{4} = \frac{(n-1)(n+4)}{4}.$$

c'est-à-dire $\Theta(n^2)$.

(4.3) **Variante de Shell**

Elle est d'une efficacité redoutable. On dit qu'un tableau T est *h-trié* si, pour tout i, le sous-tableau $[T[i], T[i+h], T[i+2h], \ldots]$ est trié. L'idée de la méthode de *Shell* est de h-trier T par insertion pour des valeurs décroissantes de h, jusqu'à $h = 1$.

On utilise l'algorithme suivant :

```
h ← n div 2
tant que h ≥ 1 faire        { h-trier le tableau T, c'est-à-dire : }
        pour i variant de h + 1 à n faire
                aux ← T[i]
                j ← i − h
                tant que (j > 0) et (T[j] > aux) faire
                        T[j + h] ← T[j]
```

```
                    j ← j − h
                fin du tant que
                T[j + h] ← aux
            fin du pour
            h ← h div 2
        fin du tant que
```

On remarque que l'ultime itération de la boucle principale "tant que $h \geq 1 \ldots$", c'est-à-dire pour $h = 1$, consiste en un tri normal par insertion du tableau T. Et pourtant cette méthode de tri est bien plus efficace que le serait un unique tri par insertion ! On peut prouver que sa complexité est au pire $\Theta(n^{\frac{3}{2}})$ et en moyenne approximativement $\Theta(n^{1,27})$ (voir [3], § 8.1).

5. Tri optimal

(5.1) Pour finir, nous présentons un algorithme de tri optimal appelé *tri fusion*. Basé sur la stratégie "diviser pour conquérir" (1.5), il consiste à couper le tableau en deux parties, trier séparément chaque partie puis reconstituer le tableau par interclassement, tout cela grâce à une procédure récursive :

```
tri_fusion(tableau T, entiers p, r) :      { trie le sous-tableau T[p..r], en supposant p ≤ r }
    si p < r alors
        q ← (p + r) div 2
        tri_fusion(T, p, q)
        tri_fusion(T, q + 1, r)
        interclasse(T, p, q, r)
    fin du si
```

où interclasse est une procédure utilisant un tableau auxiliaire U :

```
interclasse(tableau T, entiers p, q, r) :
                { trie T[p..r] en supposant p ≤ q < r et T[p..q] et T[q + 1..r] déjà triés }
    i ← p , j ← q + 1 , k ← 1
    tant que i ≤ q et j ≤ r faire
        si T[i] ≤ T[j] alors
            U[k] ← T[i] , i ← i + 1
        sinon
            U[k] ← T[j] , j ← j + 1
        fin du si
        k ← k + 1
    fin du tant que
    si i ≤ q alors
        pour j de i à q faire
            U[k] ← T[j] , k ← k + 1
        fin du pour
    sinon
        pour i de j à r faire
            U[k] ← T[i] , k ← k + 1
        fin du pour
    fin du si
    pour k de 0 à r − p faire
        T[p + k] ← U[k + 1]
    fin du pour
```

(5.2) **Etude de complexité**

Nous ne prenons en compte que les comparaisons entre éléments du tableau.

a. L'instruction de "recollement" interclasse(T, p, q, r) effectue au pire $r - p$ comparaisons : sa complexité dans le pire des cas est égale à la taille du sous-tableau traité diminuée de 1, donc linéaire. Si l'on applique

le théorème de (1.5) avec $a = b = 2$, on obtient que la complexité du tri fusion dans le pire des cas est $\Theta(n \log n)$.

b. Lorsque la taille initiale n du tableau est une puissance de 2, le tri fusion s'appelle aussi *tri dichotomique* et sa complexité dans le pire des cas se calcule directement, de la façon suivante :

Pour $h \in N$, on note u_h le nombre maximal de comparaisons nécessaires pour trier 2^h éléments, et $v_h = \frac{u_h}{2^h}$. On a $u_0 = 0$, $v_0 = 0$ et, pour $h \geq 1$:

$$u_h = 2\,u_{h-1} + 2^h - 1 \quad \text{et} \quad v_h = v_{h-1} + 1 - \frac{1}{2^h}$$

Donc :

$$v_h = \sum_{\ell=1}^{h} (v_\ell - v_{\ell-1}) = h - \frac{1}{2} \sum_{\ell=0}^{h-1} \frac{1}{2^\ell} = h - 1 + \frac{1}{2^h}$$

et :

$$u_h = (h-1)2^h + 1$$

Avec $n = 2^h$, on a finalement $u_h = (\log_2 n - 1)n + 1 \sim n \log_2 n$ quand $n \to \infty$ et on retrouve bien la complexité en $n \log n$.

(5.3) **Optimalité**

En fait, le tri fusion est un tri optimal car on peut démontrer le :

Théorème :

> *Pour les algorithmes de tri opérant par comparaisons, la complexité dans le pire des cas, comme la complexité en moyenne, est au minimum en $n \log n$.*

(pour une preuve : voir [1], chapitre 9 ou [2], chapitre 16).

Remarque.— Parmi les algorithmes de tri optimaux, l'un des plus utilisés est le *tri rapide* (ou *quicksort* : voir [1], chapitre 8 ou [2], chapitre 15). Mais il arrive que le tri rapide dégénère (complexité en n^2) alors que, comme nous l'avons vu, le tri fusion est, même dans le pire des cas, toujours en $n \log n$.

6. Méthode du simplexe

(6.1) **Programmation linéaire**

Etant donnés n nombres réels $(c_1, \ldots, c_n)$, il s'agit de trouver n réels $(x_1, \ldots x_n)$ maximisant l'expression :

$$z = c_1\,x_1 + \cdots + c_n\,x_n$$

sachant qu'on a les n contraintes :

$$x_1 \geq 0, \ldots, x_n \geq 0 \tag{2}$$

ainsi que m contraintes de la forme :

$$\begin{cases} a_{11}\,x_1 & + & \cdots & + & a_{1n}\,x_n & \leq & b_1 \\ & & & & & & \vdots \\ a_{m1}\,x_1 & + & \cdots & + & a_{mn}\,x_n & \leq & b_m \end{cases} \tag{3}$$

Un n-uplet $(x_1, \ldots, x_n)$ vérifiant les contraintes (2) et (3) s'appelle *solution admissible*. Compte tenu de l'expression affine des contraintes, l'ensemble K des solutions admissibles est un polyèdre convexe (au sens large : il peut être vide ou non borné) de $\mathbf{R}^n$ (voir [4], § 10.3).

On a alors le :

Théorème :

S'il existe une solution admissible optimale, alors il existe une solution admissible optimale telle que, parmi les $n+m$ inégalités (2) et (3), il y en a au moins n qui sont des égalités.

Pour une preuve , voir [4] théorème 10.3-2, ou [3] § 10.8.

On trouvera donc une telle solution à l'intersection de n hyperplans affines correspondant à des faces de K. La méthode consiste à parcourir les sommets du polyèdre K dans le sens des z croissants, tant que c'est possible ; on atteint ainsi une solution optimale en un nombre fini d'étapes.

Dans la suite, nous supposerons pour simplifier que le n-uplet $(0,\dots,0)$ est une solution admissible (sans quoi il peut être délicat de trouver une solution admissible initiale, s'il en existe).

(6.2) **Exemple**

Un artisan peut fabriquer deux types de produits. Chaque produit utilise trois ressources et on a les quantités suivantes :

	produit 1	produit 2	stock
ressource 1	1	3	18
ressource 2	1	1	8
ressource 3	2	1	14
bénéfice	20	30	

Comment doit-il *programmer* sa production pour faire le bénéfice maximum ?

Si l'on note x_1 le nombre de produits de type 1 et x_2 le nombre de produits de type 2, il s'agit de maximiser le gain :

$$z = 20\,x_1 + 30\,x_2$$

sous les contraintes :

$$\begin{array}{rll} x_1 & \geq 0 & (a)\\ x_2 & \geq 0 & (b)\\ x_1 + 3\,x_2 & \leq 18 & (c)\\ x_1 + x_2 & \leq 8 & (d)\\ 2\,x_1 + x_2 & \leq 14 & (e) \end{array}$$

Ce problème peut se résoudre graphiquement : voir la feuille d'illustration, section 3.1. On dessine le polyèdre K. Partant de la solution admissible $(0,0)$ correspondant à un gain nul, on passe successivement :

- au sommet $(0,6)$ correspondant à la contrainte (c) avec une égalité, et un gain $z = 180$,
- au sommet $(3,5)$ correspondant aux contraintes (c) et (d) avec des égalités, et un gain $z = 210$,

qui est l'optimum. On vérifie d'ailleurs graphiquement que $z = 210$ correspond bien à la plus grande valeur de z pour laquelle la droite d'équation $20\,x_1 + 30\,x_2 = z$ rencontre K.

(6.3) **Résolution algébrique**

Avec les notations de (6.1), on ajoute m variables (dites *d'écart*) $x_{n+1},\dots,x_{n+m}$, pour ramener les m inégalités (3) à des égalités, avec les m contraintes supplémentaires :

$$x_{n+1} \geq 0,\dots,x_{n+m} \geq 0.$$

Dans l'exemple précédent, cela conduit au :

Dictionnaire 1 :
$$\begin{cases} x_3 &= 18 - x_1 - 3\,x_2\\ x_4 &= 8 - x_1 - x_2\\ x_5 &= 14 - 2\,x_1 - x_2\\ z &= 20\,x_1 + 30\,x_2 \end{cases}$$

avec $x_1, x_2, x_3, x_4, x_5 \geq 0$. On dit que les variables x_3, x_4, x_5 sont *en base* et les variables x_1, x_2 *hors base*.

1. Choisissons une variable hors base dont le coefficient sur la dernière ligne du dictionnaire est > 0, par exemple x_2, et supposons les autres variables hors base nulles, ici $x_1 = 0$. On s'aperçoit que la contrainte

$x_3 \geq 0$ est la plus restrictive puisqu'elle implique $x_2 \leq 6$. Exprimant x_2 à l'aide de x_1 et x_3 grâce à la première équation du dictionnaire, on obtient le :

$$\textit{Dictionnaire 2 :} \quad \begin{cases} x_2 &= 6 &- \frac{1}{3}x_1 &- \frac{1}{3}x_3 \\ x_4 &= 2 &- \frac{2}{3}x_1 &+ \frac{1}{3}x_3 \\ x_5 &= 8 &- \frac{5}{3}x_1 &+ \frac{1}{3}x_3 \\ z &= 180 &+ 10\,x_1 &- 10\,x_3 \end{cases}$$

On dit qu'on a *entré x_2 dans la base* et *sorti x_3 de la base.*

2. L'étape suivante, analogue, consiste à entrer x_1 dans la base et sortir x_4, d'où le :

$$\textit{Dictionnaire 3 :} \quad \begin{cases} x_2 &= 5 &- \frac{1}{2}x_3 &+ \frac{1}{2}x_4 \\ x_1 &= 3 &+ \frac{1}{2}x_3 &- \frac{3}{2}x_4 \\ x_5 &= 3 &- \frac{1}{2}x_3 &+ \frac{5}{2}x_4 \\ z &= 210 &- 5\,x_3 &- 15\,x_4 \end{cases}$$

3. Sur la dernière ligne du dictionnaire, les coefficients en les variables hors base sont tous négatifs. Il n'est donc plus possible d'augmenter z dont le maximum est par conséquent égal à 210.

Voir la feuille d'illustration, section 3.2.

(6.4) **Algorithme du simplexe**

On représente le dictionnaire initial par un tableau T_1 à $m+1$ lignes et $1+n+m$ colonnes qui, pour l'exemple précédent, est donné par :

	x_1	x_2	x_3	x_4	x_5
18	1	3	1	0	0
8	1	1	0	1	0
14	2	1	0	0	1
z	20	30	0	0	0

tableau vérifiant la propriété :

(P) : sur chaque ligne, le premier terme est égal à la somme des autres termes multipliés par les x_i (par exemple $18 = 1 \times x_1 + 3 \times x_2 + 1 \times x_3 + 0 \times x_4 + 0 \times x_5$).

L'entrée en base de x_2 et la sortie de x_3 se traduit sur T_1 par un pivotage autour du coefficient 3 situé en ligne 1, colonne 3, pour faire apparaître en troisième colonne $(1, 0, 0, 0)$, d'où un tableau T_2. Un deuxième pivotage autour de la position $(2, 2)$ achève le calcul : voir la feuille d'illustration, section 3.3. Notons qu'à chaque pivotage, le tableau conserve évidemment la propriété (P).

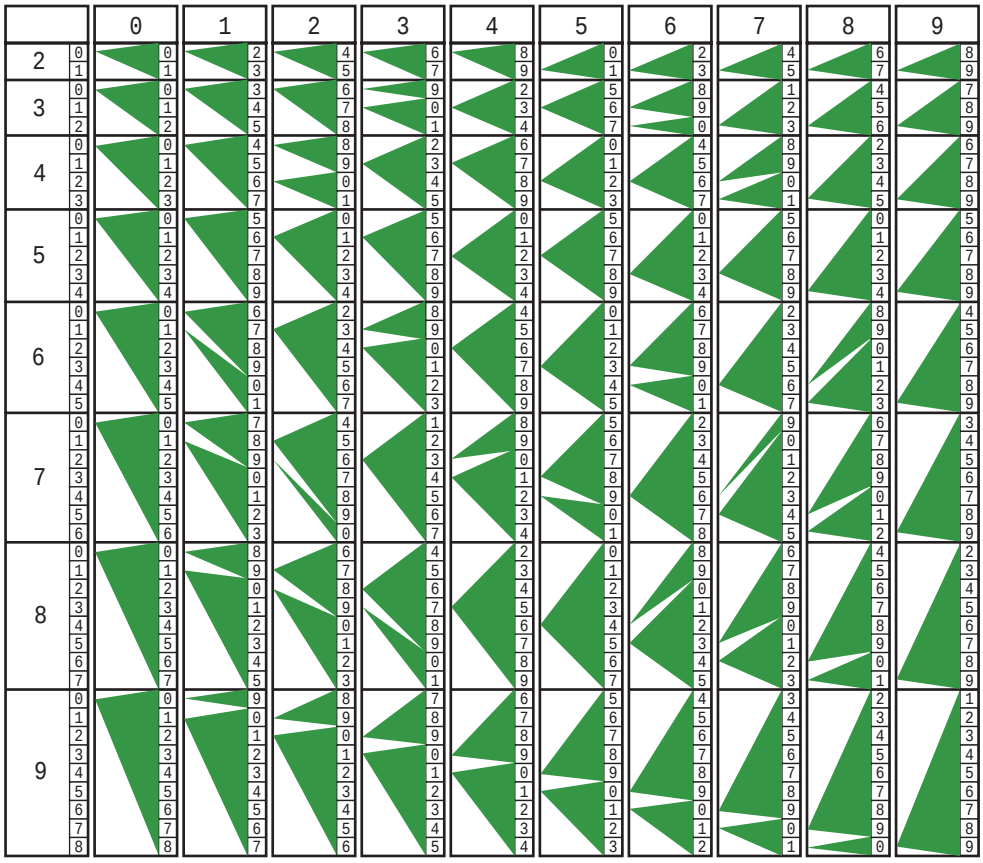
En résumé, chaque étape de l'algorithme consiste à :

- choisir une colonne pivot : colonne $s \geq 2$ où le coefficient de la dernière ligne est > 0 ; si ce n'est pas possible : fin de l'algorithme (maximum atteint),
- choisir une ligne pivot : ligne r avec $1 \leq r \leq m$, telle que :
  - le coefficient $t_{r,s}$ soit > 0 ; si ce n'est pas possible : fin de l'algorithme (fonction gain non bornée)
  - et tel que :
$$\frac{t_{r,1}}{t_{r,s}} = \min_{\substack{i=1,\ldots,m \\ t_{i,s}>0}} \frac{t_{i,1}}{t_{i,s}}$$
- utiliser le pivot $t_{r,s}$ pour faire apparaître des zéros en colonne s, sauf en position (r, s) où on fait apparaître un 1.

Références :

[1] T. Cormen, C. Leiserson et R. Rivest, *Introduction à l'algorithmique*, Dunod (1994)
[2] C. Froidevaux, M.-C. Gaudel et M. Soria, *Types de données et algorithmes*, Edisciences (1993)
[3] Press, Flannery, Teukolski & Vetterling, *Numerical recipes*, Cambridge (1989)
[4] P. G. Ciarlet, *Introduction à l'analyse numérique matricielle et à l'optimisation*, Dunod (1998)

Annexe 1 : réglettes de Genaille



Annexe 2 : réglettes trisectrices de Genaille et Lucas