Chapitre 1

Introduction

1.1 Programme, structures de données, algorithmique

L'ordinateur est un outil technologique permettant le traitement automatique de l'information. Tout programme prend en entrée de données, traite ces données et fournit en sortie de nouvelles données. L'ordinateur est donc une machine électronique permettant de transformer une information, représentée sous la forme d'une séquence de bits rangés en mémoire, en une autre information, également représentée sous la forme d'une séquence de bits rangés en mémoire.

Deux éléments caractérisent donc le traitement automatique de l'information :

— les traitements, définis sous la forme de séquences d'instructions élémentaires. Nous parlons d'**algorithmique** ;

— l'information, représentée sous la forme de **structures de données** (rangement spécifique en mémoire et codage en bits) ;

Dans les sections qui suivent nous décrivons ces deux éléments. Nous rappelons également certaines notions essentielles à la compréhension de ce cours.

1.2 Qu'est-ce que l'algorithmique ?

Définition 1 (Algorithme)**.** *Un algorithme est une série finie d'instructions élémentaires et ordonnées, constituant un modèle de résolution d'un problème.*

Le mot « algorithme » provient de la forme latine (*Algorismus*) du nom du mathématicien arabe AL-KHAREZMI ou AL-KHWĀRIZMĪ auteur –entre autres mais ce n' est pas le plus important– d'un manuel de vulgarisation sur le calcul décimal positionnel indien (v. 830) expliquant son utilisation et, surtout, la manipulation des différents algorithmes permettant de réaliser les opérations arithmétiques classiques (addition, soustraction, multiplication, division, extraction de racines carrées, règle de trois, etc.).

En dehors des mathématiques, nous sommes confrontés quotidiennement à des algorithmes : *"Une recette de cuisine"* n'est autre qu'un algorithme pour réaliser un plat. La recette nous indique les ingrédients ainsi que la façon méthodique de procéder pour arriver au résultat final (évidemment présumé être toujours "rigoureusement" le même).

Il existe deux grandes familles d'algorithmes :

— séquentiels (une instruction est exécutée après l'autre, et une seule à la fois) ;

— parallèles (des instructions peuvent être exécutées en même temps).

Dans le cadre de ce cours, nous nous intéressons essentiellement à l'algorithmique séquentielle. Dans le cas d'une recette, il s'agit souvent d'un algorithme parallèle.

Double problématique de l'algorithmique

1. **Trouver une méthode** de résolution (exacte ou approchée) du problème.
   — Soient trois nombres réels a, b et c, quelles sont les solutions de l'équation ax^2+bx+c ? (Résultat bien connu.)
   — Soient cinq nombres réels a, b, c, d et e, quelles sont les solutions de l'équation $ax^5+bx^4+cx^3+dx^2+ex+f$? (Pas de méthode générale, cf. la théorie de GALOIS.)

2. Trouver une méthode **efficace**.

Savoir résoudre un problème est une chose, le résoudre efficacement en est une autre (section 1.3).

Différences entre algorithmes et programmes

Un **programme** est la réalisation (l'implémentation) d'un algorithme (modèle de calcul ou de résolution) au moyen d'un langage donné sur une architecture donnée. Par exemple un programme écrit en langage C++, sous Unix. Il s'agit de la mise en œuvre d'un principe plus abstrait. Lors de la programmation on s'occupera explicitement de certaines notions purement techniques, comme la gestion de la mémoire (allocation dynamique), l'interface graphique, etc. qui représentent des problèmes d'implémentation souvent ignorés au niveau algorithmique. Au sein d'un logiciel l'algorithmique représente le coeur du problème, démuni de son environement technique et pratique (l'interface qui peut représenter jusqu'à 75% du logiciel et la gestion des exceptions qui peut représenter jusqu'à 20% du logiciel).

1.3 Motivation : calcul de x^n

1.3.1 Problème

Données : un entier naturel n et un réel x. On veut calculer x^n à l'aide de l'opération de multiplication notée $\times$ entre deux réels.

Moyens : Nous partons de $y_1 = x$. Nous allons construire une suite de valeurs y_1, ..., y_m telle que la valeur y_k soit obtenue par multiplication de deux puissances de x précédemment calculées : $y_k = y_u \times y_v$, avec $1 \leq u, v < k$, $k \in [2, m]$.

But : $y_m = x^n$. Le **coût** de l'algorithme sera alors de $m-1$, le nombre de multiplications faites pour obtenir le résultat recherché.

1.3.2 Algorithme naïf

$y_i = y_{i-1} \times y_1, i \in [2, n]$. Résultat : $y_n = x^n$. Coût : $m-1 = n-1$ multiplications.

Algorithme

y[1] = x
Pour $i \leftarrow 2$ **à** n **faire**
  y[i] = y[$i-1$] $\times$ y[1]
renvoyer y[n]

1.3.3 Méthode binaire

Algorithme

1. Écrire n sous forme binaire
2. Remplacer chaque :
   — « 1 » par la paire de lettres « `SX` » ;
   — « 0 » par la lettre « `S` ».
3. Éliminer la paire « `SX` » la plus à gauche.
4. Résultat : un mode de calcul de x^n où
   — `S` signifie « élever au carré » (*squaring*) ;
   — `X` signifie « multiplier par x ».
   Le tout en partant de x.

Illustration avec $n = 23$

1. $n = 10111$

   ```
   1   0   1   1   1
   ```

2. ```
   SX  S   SX  SX  SX
   ```

3. ```
       S   SX  SX  SX
   ```

4. Nous partons de x et nous utilisons la suite SSXSXSX, soit :
   x^2, x^4, x^5, x^{10}, x^{11}, x^{22}, x^{23}.
   Nous sommes donc capables de calculer x^{23} en 7 multiplications au lieu de 22 avec l'algorithme naïf.

Explication de la méthode

— Écriture binaire de n : $n = \sum_{i=0}^{i=p} a_i 2^i$.
— Plaçons nous au cours du calcul de puissances de x. Soit j le dernier bit de la représentation binaire de n qui ait été « décodé » et soit y_j le dernier résultat obtenu. Initialement, $j = p$ et $y_p = x = x^{a_p}$.
— Deux cas sont possibles pour a_{j-1} :

1. $a_{j-1} = 1$. a_{j-1} est remplacé par `SX`, nous élevons y_j au carré puis multiplions le résultat par x. Le nouveau résultat est $y_{j-1} = y_j^2 \times x$.
2. $a_{j-1} = 0$. a_{j-1} est remplacé par `S` et nous élevons simplement y_j au carré. Le nouveau résultat est $y_{j-1} = y_j^2$.

Dans tous les cas nous avons : $y_{j-1} = y_j^2 \times (x^{a_{j-1}})$.

— D'où, $y_{p-1} = y_p^2 \times (x^{a_{p-1}}) = (x^{a_p})^2 \times (x^{a_{p-1}}) = (x^{2\times a_p}) \times (x^{a_{p-1}}) = (x^{2\times a_p + a_{p-1}})$. Par récurrence, nous pouvons montrer que $y_1 = x^{\sum_{i=0}^{i=p} a_i 2^i} = x^n$...

Complexité (coût)

Note : les nombres dont la représentation binaire a exactement p chiffres forment exactement l'intervalle $[2^{p-1}, 2^p - 1]$.

Nombres de chiffres dans l'écriture binaire de n : $1 + [\log_2 n]$. Notons $\nu(n)$ le nombre de « 1 » dans l'écriture binaire de n. Nombre d'opérations effectuées :

— $(1 + [\log_2 n]) - 1$ élévations au carré (ne pas oublier l'étape 3) ;
— $\nu(n) - 1$ multiplications par x (ne pas oublier l'étape 3).

Soit en tout $T(n) = [\log_2 n] + \nu(n) - 1$ multiplications. Trivialement, $1 \leq \nu(n) \leq [\log_2 n]$ et $[\log_2 n] \leq T(n) \leq 2[\log_2 n]$.

Pour $n = 1000$, l'algorithme trivial effectue 999 multiplications, et la méthode binaire moins de 20.

Historique

Cette méthode a été présentée avant 200 avant J.C. en Inde, mais il semblerait qu'il ait fallu attendre un millénaire avant que cette méthode ne soit connue en dehors de l'Inde *p. 441 de "Donald E. Knuth", Seminumerical Algorithms,Addison Wesley,1969,The Art of Computer Programming, Volume 2*.

Peut-on faire mieux ?

Prenons le cas $n = 15$.

1. $n = 1111$

   ```
   1   1   1   1
   ```

2. ```
   SX  SX  SX  SX
   ```

3. ```
       SX  SX  SX
   ```

4. Nous partons de x et nous obtenons successivement : x^2, x^3, x^6, x^7, x^{14}, x^{15}. Nous sommes donc capables de calculer x^{15} en 6 multiplications.

Autre schéma de calcul : x^2, x^3, x^6, x^{12}, $x^{15} = x^{12} \times x^3$. Nous obtenons ainsi x^{15} en 5 multiplications et la méthode binaire n'est donc pas *optimale* (c'est-à-dire que l'on peut faire mieux).

Peut-on faire encore mieux ?

en fait oui..., mais nous n'allons pas aborder ici tous ces algorithmes qui dépassent le cadre de cette introduction.

D'après *"Thomas Cormen, Charles Leiserson and Ronald Rivest", "Introduction à l'algorithmique",Dunod,1994* : « Un bon algorithme est comme un couteau tranchant —il fait exactement ce que l'on attend de lui, avec un minimum d'efforts. L'emploi d'un mauvais algorithme pour résoudre un problème revient à essayer de couper un steak avec un tournevis : vous finirez sans doute par obtenir un résultat digeste, mais vous accomplirez beaucoup plus d'efforts que nécessaire, et le résultat aura peu de chances d'être esthétiquement satisfaisant. »

1.3.4 La notion de complexité

Définitions

Définition 2 (Complexité)**.** *La complexité d'un algorithme est la mesure du nombre d'opérations fondamentales qu'il effectue sur un jeu de données. La complexité est exprimée comme une fonction de la taille du jeu de données.*

Nous notons D_n l'ensemble des données de taille n et $T(d)$ le coût de l'algorithme sur une donnée d.

Complexité au meilleur : $T_{\min}(n) = \min_{d \in D_n} C(d)$. C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à n. C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille n.

Complexité au pire : $T_{\max}(n) = \max_{d \in D_n} C(d)$. C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à n.

Avantage : il s'agit d'un maximum, et l'algorithme finira donc toujours avant d'avoir effectué $T_{\max}(n)$ opérations.

Inconvénient : cette complexité peut ne pas refléter le comportement « usuel » de l'algorithme, le pire cas ne pouvant se produire que rarement.

Complexité en moyenne : $T_{\text{moy}}(n) = \frac{\sum_{d \in D_n} C(d)}{|D_n|}$. C'est l'espérance mathématique des complexités de l'algorithme sur des jeux de données de taille n où ces jeux de données sont exprimés dans un espace probabilisé (concrètement, on se donne la probabilité d'apparition de chacun des jeux de données).

Avantage : reflète le comportement « général » de l'algorithme si les cas extrêmes sont rares ou si la complexité varie peu en fonction des données.

Inconvénient : la complexité en pratique sur un jeu de données particulier peut être nettement plus importante que la complexité en moyenne, dans ce cas la complexité en moyenne ne donnera pas une bonne indication du comportement de l'algorithme.

En pratique, nous ne nous intéresserons qu'à la complexité au pire et à la complexité en moyenne.

Définition 3 (Efficacité)**.** *Un algorithme A est dit plus **efficace**, respectivement au pire ou en moyenne, qu'un algorithme B de sa classe, si la complexité de A, respectivement au pire ou en moyenne, est inférieure à celle B.*

Définition 4 (Optimalité)**.** *Un algorithme est dit **optimal** si sa complexité est la complexité minimale parmi tous les algorithmes de sa classe.*

Nous nous intéresserons quasi exclusivement à la *complexité en temps* des algorithmes. Il est parfois intéressant de s'intéresser à d'autres caractéristiques, comme la *complexité en espace* (taille de l'espace mémoire utilisé), la largeur de bande passante requise, etc.

Notations de Landau

Quand nous calculons la complexité d'un algorithme, nous ne calculons généralement pas sa complexité exacte, mais son ordre de grandeur. Pour ce faire, nous avons besoin de notations asymptotiques.

$$
\begin{array}{lcll}
\mathbf{O} & : & f = O(g) & \Leftrightarrow \quad \exists n_0, \exists c \geq 0, \forall n \geq n_0, f(n) \leq c \times g(n) \\
\mathbf{\Omega} & : & f = \Omega(g) & \Leftrightarrow \quad g = O(f) \\
\mathbf{o} & : & f = o(g) & \Leftrightarrow \quad \forall c > 0, \exists n_0, \forall n \geq n_0, f(n) \leq c \times g(n) \\
\mathbf{\Theta} & : & f = \Theta(g) & \Leftrightarrow \quad f = O(g) \text{ et } g = O(f)
\end{array}
$$

1.3.5 Modèle de machine

Pour que le résultat de l'analyse d'un algorithme soit pertinent, il faut avoir un modèle de la machine sur laquelle l'algorithme sera implémenté (sous forme de programme). On prendra comme référence un modèle de **machine à accès aléatoire (RAM)** et à processeur unique, où les instructions sont exécutées l'une après l'autre, sans opérations simultanées (algorithmique séquentielle).

1.4 Structures de données élémentaires

En informatique, la notion **d'ensemble** joue un rôle prépondérant. Il s'agit d'un outil fondamental permettant d'opérer un regroupement et une classification d'objets sur lesquels doit s'exécuter un traitement. Il existe plusieurs manières de représenter la notion **d'ensemble** (qui n'est pas forcément la notion mathématique d'ensemble). Dans un ensemble un élément existe ou n'existe pas (il fait partie ou non de l'ensemble). En informatique on parlera plus de **collection** : un même élément peut avoir plusieurs occurences dans une collection. Il n'existe pas une représentation qui soit « meilleure » que les autres dans l'absolu : pour un problème donné, la meilleure représentation sera celle qui permettra de concevoir le « meilleur » algorithme, c'est-à-dire celui le plus esthétique et de moindre complexité. On parle de*collections dynamiques* car les collections seront rarement figées dans le temps. Généralement le choix, en terme représentation par une structure de données, est celui d'opter pour une représentation contigüe ou pour une représentation chaînée, ou bien pour un mélange des deux.

Chaque élément d'une collection pourra comporter plusieurs *champs* qui peuvent être examinés dès lors que l'on possède un *pointeur* —ou une *référence* si on préfère utiliser une terminologie plus proche de *Java* que de *C*— sur cet élément. Certaines collections supposent que l'un des champs de l'objet contient une **clé** unique servant d'identifiant.

Les collections que nous allons considérer supportent potentiellement une série basique d'opérations :

— RECHERCHE(S, k) : étant donné un ensemble S et une clé k, le résultat de cette requête est un "pointeur" sur un élément de S de clé k, s'il en existe un, et la valeur NIL sinon —NIL étant un pointeur ou une référence sur « rien ».
— INSERTION(S, x) : ajoute à l'ensemble S l'élément pointé par x.
— SUPPRESSION(S, x) : supprime de l'ensemble S son élément pointé par x (si l'on souhaite supprimer un élément dont on ne connaît que la clé k, il suffit de récupérer un pointeur sur cet élément via un appel à RECHERCHE(S, k)).

Si l'ensemble des clés, ou l'ensemble lui-même, est totalement ordonné, d'autres opérations sont possibles :

— MINIMUM(S) : renvoie l'élément de S le plus petit si S est non vide.
— MAXIMUM(S) : renvoie l'élément de S le plus grand si S est non vide.
— SUIVANT(S, x) : renvoie, si celui-ci existe, l'élément de S immédiatement plus grand que l'élément de S pointé par x, et NIL dans le cas contraire.
— PRÉCÉDENT(S, x) : renvoie, si celui-ci existe, l'élément de S immédiatement plus petit que l'élément de S pointé par x, et NIL dans le cas contraire.
— PREMIER(S) : renvoie l'élément de S de clé minimale.
— DERNIER(S) : renvoie l'élément de S de clé maximale.
— SUCCESSEUR(S, x) : renvoie, si celui-ci existe, l'élément de S ayant une clé immédiatement plus grande que la clé de l'élément de S pointé par x, et NIL dans le cas contraire.
— PRÉDÉCESSEUR(S, x) : renvoie, si celui-ci existe, l'élément de S ayant une clé immédiatement plus petite que l'élément de S pointé par x, et NIL dans le cas contraire.

En dehors de cela il est possible de définir d'autres opérations, comme le cardinal (nombre d'éléments), le test de vacuité, etc.

Certaines collections sont des *arrangements linéaires* : dans ce cas, la clé représente souvent le rang de l'élément dans cet arrangement. Il est important de ne pas confondre collection et tableau, ni collection et liste chaînée. Le tableau et la liste chaînée ne sont que des outils d'implantation informatique servant à réaliser des collections. Rappelons par la suite trois types de collections ayant des caractéristiques particulières : la pile, la file et la liste.

1.4.1 Piles

Définition 5 (Pile)**.** *Une pile est une structure de données mettant en œuvre le principe « dernier entré, premier sorti » (*LIFO : Last-In, First-Out *en anglais).*

L'élément ôté de l'ensemble par l'opération SUPPRESSION est spécifié à l'avance (et donc cette opération ne prend alors que l'ensemble comme argument) : l'élément supprimé est celui le plus récemment inséré. L'opération INSERTION dans une pile est communément appelée EMPILER, et l'opération SUPPRESSION, DÉPILER.

Il est facile d'implémenter une pile au moyen d'un tableau. La seule difficulté dans cette implémentation est la gestion des débordements de pile qui interviennent quand on tente d'effecteur l'opération DÉPILER sur une pile vide et l'opération EMPILER sur un tableau codant la pile qui est déjà plein. Ce dernier problème n'apparaît pas lorsque l'on implémente les piles au moyen d'une structure de données dont la taille n'est pas fixée *a priori* (comme une liste chaînée).

PILE-VIDE(P)
  si *sommet*(P)=0 **alors renvoyer** VRAI
  sinon renvoyer FAUX

EMPILER(P, x)
  si *sommet*(P) = *longueur*(P) **alors erreur** « débordement positif »
  sinon *sommet*(P) $\leftarrow$ *sommet*(P)+1
  $P[sommet(P)] \leftarrow x$

DÉPILER(P)
  si PILE-VIDE(P) **alors erreur** « débordement négatif »
  sinon *sommet*(P) $\leftarrow$ *sommet*(P)-1
  renvoyer $P[sommet(P)+1]$

FIGURE 1.1 – Algorithmes de manipulation des piles implémentées par des tableaux.

1.4.2 Files

Définition 6 (File)**.** *Une file est une structure de données mettant en œuvre le principe « premier entré, premier sorti » (*FIFO : First-In, First-Out *en anglais).*

L'élément ôté de l'ensemble par l'opération SUPPRESSION est spécifié à l'avance (et donc cette opération ne prend alors que l'ensemble comme argument) : l'élément supprimé est celui qui est resté le plus longtemps dans la file. Une file se comporte exactement comme une file d'attente de la vie courante.

On peut implémenter les files au moyen de tableaux. On peut implémenter des files à $n-1$ éléments au moyen d'un tableau à n éléments et de deux attributs :

— *tête*(F) qui indexe (ou pointe) vers la tête de la file ;
— *queue*(F) qui indexe le prochain emplacement où sera inséré un élément nouveau.

Les éléments de la file se trouvent donc aux emplacements *tête*(F), *tête*(F)+1, ..., *queue*(F)-1 (modulo n). Quand *tête*(F) = *queue*(F), la liste est vide. La seule difficulté d'implémentation est la gestion des débordements de file qui interviennent quand on tente d'effectuer l'opération SUPPRESSION sur une pile vide et l'opération INSERTION sur un tableau codant la file qui est déjà plein. Ce dernier problème n'apparaît pas lorsque l'on implémente les files au moyen d'une structure de donnée dont la taille n'est pas fixée *a priori* (comme une liste doublement chaînée).

FILE-VIDE(F)
  si *tête*(F)=*queue*(F) **alors renvoyer** VRAI
  sinon renvoyer FAUX

INSERTION(F, x)
  si *queue*(F) + 1 (modulo n) = *tête*(F) **alors erreur** « débordement positif »
  sinon $F[queue(F)] \leftarrow x$
  $queue(F) \leftarrow queue(F)+1$

SUPPRESSION(F)
  si FILE-VIDE(F) **alors erreur** « débordement négatif »
  sinon $tête(F) \leftarrow tête(F)+1$
  renvoyer $F[tête(F)-1]$

FIGURE 1.2 – Algorithmes de manipulation des files implémentées par des tableaux.

1.4.3 Listes

Définition 7 (Liste)**.** *Une liste est une structure de données dans laquelle les objets sont arrangés linéairement. La clé représente la position ou le rang d'un élément dans cet arrangement.*

Une liste peut être réalisée par un chaînage. Dans ce cas, chaque élément de la liste, outre le champ *clé*, contient un champ *successeur* qui est un pointeur sur l'élément suivant dans la liste. Si le champ *successeur* d'un élément vaut NIL, cet élément n'a pas de successeur et est donc le dernier élément ou la **queue** de la liste. Le premier élément de la liste est appelé la **tête** de la liste. Une liste L représentée sous la forme d'un chaînage est manipulée via un pointeur vers son premier élément, que l'on notera TÊTE(L). Si TÊTE(L) vaut NIL, la liste est vide.

Une liste chaînée peut prendre plusieurs formes :

— **Liste doublement chaînée** : en plus du champ *successeur*, chaque élément contient un champ *prédécesseur* qui est un pointeur sur l'élément précédant dans la liste. Si le champ *prédécesseur* d'un élément vaut NIL, cet élément n'a pas de prédécesseur et est donc le premir élément ou la **tête** de la liste. Une liste qui n'est pas doublement chaînée est dite **simplement chaînée**.

— **Triée** ou **non triée** : suivant que l'ordre linéaire des éléments dans la liste correspond ou non à l'ordre linéaire des clés de ces éléments.

— **Circulaire** : si le champ *précécesseur* de la tête de la liste pointe sur la queue, et si le champ *successeur* de la queue pointe sur la tête. La liste est alors vue comme un anneau.

La liste peut également être représentée par un tableau. Dans ce cas, la clé correspond à l'indice de l'élément dans le tableau. L'insertion et la suppression implique des opérations de décalage vers la gauche ou vers la droite.

1.4.4 Comparaison entre tableaux et listes chaînées

Aucune structure de données n'est parfaite, chacune a ses avantages et ses inconvénients. La figure 1.3 présente un comparatif des listes simplement chaînées, doublement chaînées et des tableaux, triés ou non, sur des opérations élémentaires. Les complexités indiquées sont celles du *pire cas*. Suivant les opérations que nous aurons à effectuer, et suivant leurs fréquences relatives, nous choisirons l'une ou l'autre de ces structures de données.

	liste chaînée simple non triée	liste chaînée simple triée	liste chaînée double non triée	liste chaînée double triée	tableau non trié	tableau trié
RECHERCHE(L, k)	$\Theta(n)$ [a]	$\Theta(n)$[a]	$\Theta(n)$[a]	$\Theta(n)$[a]	$\Theta(1)$ [b]	$\Theta(1)$[b]
INSERTION(L, x)	$\Theta(1)$	$\Theta(n)$ [e]	$\Theta(1)$	$\Theta(n)$[e]	$\Theta(n)$ [c] ou erreur[f]	$\Theta(n)$ [d] $\Theta(n)$[c] ou erreur[f]
SUPPRESSION(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$ [g]	$\Theta(n)$[g]
SUCCESSEUR(L, x) [h]	$\Theta(n)$ [i]	$\Theta(1)$	$\Theta(n)$[i]	$\Theta(1)$	$\Theta(n)$[i]	$\Theta(1)$
PRÉDÉCESSEUR(L, x) [h]	$\Theta(n)$[i]	$\Theta(n)$ [j]	$\Theta(n)$[i]	$\Theta(1)$	$\Theta(n)$[i]	$\Theta(1)$
MINIMUM(L)	$\Theta(n)$[i]	$\Theta(1)$	$\Theta(n)$[i]	$\Theta(1)$	$\Theta(n)$[i]	$\Theta(1)$
MAXIMUM(L)	$\Theta(n)$[i]	$\Theta(n)$ [k]	$\Theta(n)$[i]	$\Theta(n)$[k]	$\Theta(n)$[i]	$\Theta(1)$

a. Dans le pire cas il faut parcourir tous les éléments pour se rendre compte que la clef n'était pas dans l'ensemble.

b. La clé étant l'indice de l'élément dans le tableau.

c. Dans le pire cas, il faut allouer un nouveau tableau et recopier tous les éléments de l'ancien tableau dans le nouveau.

d. Dans le pire cas, l'insertion a lieu dans la première cas du tableau, et il faut décaler tous les éléments déjà présents.

e. Au pire, l'insertion a lieu en fin de liste.

f. Au cas où l'on veut effectuer une insertion dans un tableau déjà plein et qu'il n'est pas possible d'effectuer une allocation dynamique de tableau, comme en FORTRAN 77 ou en PASCAL.

g. Dans le pire cas on supprime le premier élément du tableau et il faut décaler tous les autres éléments.

h. Au sens de l'ordre sur la valeur des clés.

i. Complexité de la recherche du maximum (ou du minimum) dans un ensemble à n éléments...

j. Complexité de la recherche du maximum dans un ensemble à n éléments... car il faut entreprendre la recherche du prédécesseur depuis le début de la liste.

k. Il faut parcourir la liste en entier pour trouver son dernier élément.

FIGURE 1.3 – Efficacités respectives des listes chaînées et des tableaux.