Mathématiques pour l'informatique

(option de 1e année du DEUG MIAS)

D. Montlouis-Calixte, M. Hasler

Table des matières

1 Présentation du cours **3**
- 1.1 Objectif . . . 3
- 1.2 Durée et déroulement . . . 3

2 Rappels d'analyse **4**
- 2.1 Comparaison asymptotique de suites . . . 4
  - 2.1.1 Domination . . . 4
  - 2.1.2 Équivalence grossière . . . 4
  - 2.1.3 Prépondérance (ou négligeabilité) . . . 5
  - 2.1.4 Équivalence (asymptotique) . . . 5
  - 2.1.5 Cas des fonctions . . . 6
- 2.2 Somme partielle de suites . . . 6
  - 2.2.1 Comparaison avec une intégrale . . . 6
  - 2.2.2 Comparaison avec une suite équivalente . . . 6
- 2.3 Terme général et somme partielle de suites récurrentes linéaires . 7
  - 2.3.1 Suite arithmétique . . . 7
  - 2.3.2 Suite géométrique . . . 7
  - 2.3.3 Suite $u_{n+1} = a\,u_n + b$ . . . 7
  - 2.3.4 Sous- et sur-suites . . . 8
- 2.4 Exercices . . . 8

3 Notion d'algorithme **9**
- 3.1 Variables et assertions . . . 9
  - 3.1.1 Variables . . . 9
  - 3.1.2 Assertions . . . 9
- 3.2 Instructions, pré- et post-conditions . . . 10
- 3.3 Structures de contrôle . . . 11
  - 3.3.1 Séquence . . . 11
  - 3.3.2 Alternative (si ... alors ...) . . . 12
  - 3.3.3 Boucle (tant que) . . . 12
  - 3.3.4 Autres structures de contrôle . . . 13
- 3.4 Algorithme . . . 13
  - 3.4.1 Définition . . . 13
  - 3.4.2 Exemple : Euclide (calcul du PGCD) . . . 14
  - 3.4.3 Structure fréquente d'algorithmes . . . 14

4 Preuve d'un algorithme **15**
4.1 Correction partielle 15
4.2 Correction (terminaison) 15
4.3 Invariant de boucle 16
4.4 Preuve de correction partielle 16
4.5 Preuve de la terminaison 16
4.6 Exemple de preuve : tri par insertion 17

5 Notion de complexité **18**
5.1 Introduction 18
5.2 Complexité temporelle et spatiale 18
5.2.1 Complexité temporelle 18
5.2.2 Complexité spatiale 19
5.3 Complexité moyenne, au pire et meilleur des cas 19
5.4 Classes de complexité 20

6 Exemples de calculs de complexité **21**
6.1 Recherche linéaire dans un tableau 21
6.2 Algorithme dichotomique 21
6.3 Recherche dichotomique dans un tableau trié 23
6.4 Exponentiation entière 23
6.5 Tri par ségmentation 25
6.5.1 Ségmentation d'un tableau 25
6.5.2 Tri par segmentation 26
6.6 Méthode "diviser pour régner" 26

1 Présentation du cours

1.1 Objectif

Ce cours a pour but d'introduire des notions de mathématiques s'appliquant à l'informatique, en particulier l'évaluation de la complexité temporelle (coût en temps d'execution) et spatiale (taille de mémoire nécessaire) d'algorithmes.

Ce sujet ne peut être abordé sans

- d'une part, des connaissances de base concernant la comparaison asymptotique de suites numériques, et
- d'autre part, les notions d'algorithme, correction (partielle) et terminaison, donc de techniques élémentaires de preuve d'algorithme.

Ces deux sujets sont donc brièvement rappelés resp. introduits dans une présentation sommaire.

Le cours ne vise pas à maximiser la quantité de savoir transmis, mais plutôt de faire passer une démarche scientifique dans l'étude des problèmes. En particulier, l'accent est mis sur la présentation de méthodes générales et la mise en œuvre d'outils pour résoudre des problèmes du type considéré.

Enfin, même si ce cours cherche à être le plus autosuffisant possible, il s'appuie néanmoins sur le bénéfice de formations antérieures en informatique et mathématiques.

1.2 Durée et déroulement

Cette unité d'enseignement correspond à un volume horaire de 30h, et sera divisé en 12 séances de cours d'une heure et demie, et 12 séances de TD d'une heure. A chaque séance de TD, trois exercices au maximum, portant sur le cours précédent, seront résolus par les étudiants en travail dirigé par l'enseignant.

2 Rappels d'analyse

2.1 Comparaison asymptotique de suites

On s'intéresse au comportement de suites infinies, c'est-à-dire de fonctions définies sur une partie infinie de $\mathbb{N}$. Ces propriétés ne dépendent pas des premiers termes de la suite. On dira qu'une propriété est vérifiée à partir d'un certain rang, si elle est vérifiée pour tous les $n \in \mathbb{N}$ supérieurs à un certain $n_0 \in \mathbb{N}$. Pour exprimer ceci, on écrit parfois aussi "(∞)" en dessous d'une relation (par exemple, $u_n \underset{(\infty)}{\leq} v_n$).

Parfois il sera utile de faire le changement de variable $t = 1/n$ pour se ramener à l'étude d'une fonction définie au voisinage de 0, et de son comportement lorsque $t > 0$ tend vers 0, au moyen de techniques telles que les developpements limités, par exemple.

2.1.1 Domination

Dfinition 2.1 *Soient $u = (u_n)$ et $v = (v_n)$ deux suites. On dit que u est dominée par v si, et seulement si, il existe une constante c telle qu'à partir d'un certain rang, on ait $|u_n| \leq c\,|v_n|$.*
Dans ce cas, on écrit $u \preceq v$ (notation de Hardy) ou $u = O(v)$ (« u est un grand O de v » ; notation de Landau). La relation réciproque est : v domine u, ce qui est parfois noté $v = \Omega(u)$.

Proposition 2.2 *Cette relation est reflexive ($u = O(u)$ pour toute suite u) et transitive ($u = O(v)$ et $v = O(w)$ implique $u = O(w)$). Elle est compatible avec la multiplication, et avec l'addition si les suites du membre de droite sont à termes positives.*

(Cela signifie que si u, v, w, z sont des suites à termes positifs, alors $u = O(v)$ et $w = O(z)$ (en particulier $w = z$) implique $(u + w) = O(v + z)$. Cependant, si les suites ne sont pas à termes positifs, les termes de v et z peuvent se compenser, et la relation peut ne pas être vérifiée pour la somme (par exemple, si $z = -v$, auquel cas $v+z = o$ (la suite nulle), qui ne domine que la suite nulle uniquement. Dans le cas de suites réelles, on pourrait omettre la valeur absolue dans le membre de droite, auquel cas v est forcément à valeurs positifs. Cependant, il est plus naturel de mettre des valeurs absolues partout, en vue de la généralisation aux suites dans des espaces vectoriels normés.)

2.1.2 Équivalence grossière

Dfinition 2.3 *Soient $u = (u_n)$ et $v = (v_n)$ deux suites. On dit que u est grossièrement équivalente à v si, et seulement si, il existent des constantes strictement positives a, b telles qu'on ait $a\,|u_n| \leq |u_n| \leq b\,|v_n|$ à partir d'un certain rang. Dans ce cas, on écrit $u = \Theta(v)$.*

Remarque 2.4 *Deux suites sont grossièrement équivalentes ssi (=si, et seulement si) chacune domine l'autre, c-à-d.* $u = O(v)$ *et* $v = O(u)$.

Proposition 2.5 *C'est une relation d'équivalence, c-à-d. elle est reflexive, transitive et symétrique (*$u = \Theta(v)$ *implique* $v = \Theta(u)$*).*

2.1.3 Prépondérance (ou négligeabilité)

Dfinition 2.6 *Soient* $u = (u_n)$ *et* $v = (v_n)$ *deux suites. On dit que* u *est négligeable devant* v*, ou que* v *est prépondérante devant* u*, si, et seulement si, pour tout* $\varepsilon > 0$*, on ait* $|u_n| \leq \varepsilon\, |v_n|$ *à partir d'un certain rang. Dans ce cas, on écrit* $u \ll v$ *(notation de Hardy) ou* $u = o(v)$ *(« u est un petit o de v » ; notation de Landau).*

Proposition 2.7 *Cette relation est transitive (*$u = o(v)$ *et* $v = o(w)$ *implique* $u = o(w)$*). Elle est compatible avec la multiplication, et avec l'addition si les suites du membre de droite sont à termes positives. Clairement,* $u = o(v) \Rightarrow u = O(v)$.

Proposition 2.8 *On a* $u = o(v)$ *ssi il existe une suite* (ε_n) *qui converge vers* 0*, telle que* $u_n = \varepsilon_n\, v_n$ *à partir d'un certain rang.*

Remarque 2.9 *Si la suite* v *est à valeurs non-nulles à partir d'un certain rang, alors* $u = o(v)$ *ssi* $\lim u_n/v_n = 0$.

2.1.4 Équivalence (asymptotique)

Dfinition 2.10 *Soient* $u = (u_n)$ *et* $v = (v_n)$ *deux suites. On dit que* u *est (asymptotiquement) équivalente à* v *si, et seulement si,* $(u - v)$ *est négligeable devant* u*. Dans ce cas, on écrit* $u \sim v$.

Proposition 2.11 *C'est une relation d'équivalence, c-à-d. elle est reflexive, transitive et symétrique. Evidemment,* $u \sim v \Rightarrow u = \Theta(v)$.

Remarque 2.12 *Si la suite* v *est à valeurs non-nulles à partir d'un certain rang, alors* $u \sim v$ *ssi* $\lim u_n/v_n = 1$.

Pour décrire le comportement asymptotique d'une suite, il suffit souvent de connaître une suite equivalente plus simple. A un coefficient près, cette suite pourra presque toujours être choisie parmi l'*echelle* $\left\{e^{\alpha\, n}\, n^{\beta}\, (\log n)^{\gamma}\right\}_{\alpha,\beta,\gamma\in\mathbb{R}}$, chacun des paramètres α, β, γ l'emportant sur le suivant au point de vue de la croissance. (Plus rarement, on rencontre une croissance encore plus forte, par exemple de type $n! \sim \sqrt{2\,\pi\, n}(\frac{n}{e})^n$ (formule de Stirling), ou encore plus faible, comme $\log(\log(n))$.)

Exercice 2.1.1 *Donner des équivalents simples de* $\frac{n^2+n^3-1}{n-2\,n^2+1}$ *et* $\frac{n\log(n)+1}{\sqrt{n}}$*, pour ensuite comparer leur comportement asymptotique.*

Exercice 2.1.2 *Montrer que* $v_n = \frac{1}{2\,n}$ *est un équivalent de* $u_n = \sqrt{n^2+1} - n$.

2.1.5 Cas des fonctions

Lors de l'étude de suites, on omet souvent de préciser qu'une certaine relation est vérifiée "au voisinage de l'infini", c-à-d. à partir d'un certain rang. Lorsqu'on généralise ce qui précède à l'étude du comportement de fonctions au voisinage d'un point $a \in \overline{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$ (par exemple, lorsqu'on fait un changement de variable $t = 1/n$), il convient de toujours préciser ce point a, soit en ajoutant $(t \to a)$ après la relation, ou au moins (a) en dessous du symbole de la relation.

Exemple 2.13 *Soit $f : x \mapsto \frac{1}{x}$ et $g : x \mapsto e^x$. Alors $f \underset{(\infty)}{=} o(g)$ mais $g \underset{(0)}{=} o(f)$.*

2.2 Somme partielle de suites

2.2.1 Comparaison avec une intégrale

Dans la suite, on compare la somme partielle, $S_p^n(u) = u_p + u_{p+1} + \cdots + u_n$, d'une suite u définie par $u_n = f(n)$, à une intégrale de la forme $\int_p^n f(x)\,\mathrm{d}x$. En visualisant les sommes de Riemann de f pour la subdivision $\{x_i = p + i\}_{0 \le i \le n}$ de l'intervalle $[p, n+1]$ (aire des rectangles de base $[x_i, x_{i+1}]$ et de hauteur $f(x_i)$ resp. $f(x_{i+1})$), on voit que

$$\text{si } f \nearrow, \text{ alors } u_p + u_{p+1} + \cdots + u_n \le \int_p^{n+1} f(x)\,\mathrm{d}x \le u_{p+1} + \cdots + u_{n+1}\ ,$$

$$\text{si } f \searrow, \text{ alors } u_p + u_{p+1} + \cdots + u_n \ge \int_p^{n+1} f(x)\,\mathrm{d}x \ge u_{p+1} + \cdots + u_{n+1}\ .$$

En changeant $n+1$ en n, on obtient :

Proposition 2.14 *On considère une suite $u_n = f(n)$, où f est une fonction **croissante** sur $[p, \infty[$. Dans ce cas, on a*

$$u_p + \int_p^n f(x)\,\mathrm{d}x \le u_p + u_{p+1} + \cdots + u_n \le \int_p^n f(x)\,\mathrm{d}x + u_n\ ,$$

*Si f est **décroissante**, on a les mêmes relations avec $\ge$ au lieu de $\le$. Si u_p et u_n sont négligeables devant $\int_p^n f(x)\,\mathrm{d}x$ (lorsque $n \to \infty$; donc seulement si $\int_p^n f(x)\,\mathrm{d}x$ diverge), alors $u_p + \cdots + u_n \underset{n\to\infty}{\sim} \int_p^n f(x)\,\mathrm{d}x$.*

Exercice 2.2.1 *Encadrer $r_n = \sqrt{0} + \sqrt{1} + \sqrt{2} + \cdots + \sqrt{n}$ et en déduire un équivalent.*

Exercice 2.2.2 *Donner un équivalent simple de $s_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$, puis de $v_n = \ln 2 + \ln 3 + \cdots + \ln(n)$.*

2.2.2 Comparaison avec une suite équivalente

Theorme 2.15 *Si $u_n \sim v_n$ et $\lim \sum_{k=p}^n u_n = \infty$, alors $\sum_{k=p}^n u_n \sim \sum_{k=p}^n v_n$.*

Exercice 2.2.3 *Donner un équivalent de $s_n = \sin 1 + \sin\frac{1}{2} + \sin\frac{1}{3} + \cdots + \sin\frac{1}{n}$.*

2.3 Terme général et somme partielle de suites récurrentes linéaires

Nous ne considérons ici que des suites récurrentes de premier order, donc de la forme $u_{n+1} = f(u_n)$.

2.3.1 Suite arithmétique

Proposition 2.16 *Si u est une suite arithmétique de raison r, c-à-d. $u_{n+1} = u_n + r$, alors*

$$u_n = u_p + (n-p)\,r \quad (= u_0 + n\,r)$$

et toute somme partielle est égale au nombre de termes multiplié par la valeur moyenne, soit

$$S_p^n(u) = u_p + u_{p+1} + \cdots + u_n = \frac{u_p + u_n}{2}(n-p+1) \underset{n\to\infty}{\sim} \frac{r}{2}\,n^2 \ .$$

Remarque 2.17 *On peut considérer en particulier le cas $p = 1$, et retrouver la formule pour $\sum_{k=1}^n k$, à connaître par cœur !*

2.3.2 Suite géométrique

Proposition 2.18 *Si u est une suite géométrique de raison q, c-à-d. $u_{n+1} = q\,u_n$, alors*

$$u_n = u_p\,q^{n-p}$$

et la somme partielle est égale à

$$S_p^n(u) = u_p + u_{p+1} + \cdots + u_n = \frac{u_p - q\,u_n}{1-q} \ .$$

(Pour $q = 1$, on a une suite constante, donc $S_p^n(u) = (n-p+1)\ u_p$.)
Pour $|q| < 1$, u_n tend vers zero ; pour $|q| > 1$ on a $S_p^n(u) \sim \frac{u_p}{q-1}\,q^{n-p+1}$ $(n \to \infty)$.

Exercice 2.3.1 *Retrouver la formule pour $\sum_{k=p}^n q^k$, à connaître par cœur.*

2.3.3 Suite $u_{n+1} = a\,u_n + b$

A une telle suite on peut associer l'application affine $f(x) = a\,x + b$. Pour $a = 1$, on retrouve une suite arithmétique, pour laquelle $u_n = u_0 + n\,b$. Considèrons donc dans la suite le cas $a \neq 1$. L'application f possède alors un unique point fixe, $\ell = \frac{b}{1-a}$, et la suite $v_n = u_n - \ell$ est une suite géométrique de raison $q = a$. Ainsi, u_n est de terme général

$$u_n = \ell + (u_p - \ell)\,a^{n-p} \ .$$

2.3.4 Sous- et sur-suites

Considérons une suite vérifiant $u_{n+1} \leq f(u_n)$ pour tout n (resp. $u_{n+1} \geq f(u_n)$ pour tout n), où f est une fonction monotone qu'on supposera **croissante** (dans le cas contraire, il suffit de considérer la suite $u'_n = -u_n$). Si v_n est la suite vérifiant $v_p = u_p$ et $v_{n+1} = f(v_n)$ pour tout $n \geq p$, alors on appelle v_n une sur-suite (resp. sous-suite) de (u_n), et on a $u_n \leq v_n$ (resp. $u_n \geq v_n$) pour tout $n \geq p$.

2.4 Exercices

Exercice 2.4.1 *On considère les suites*

$$u_n = 1 \times 2^1 + 2 \times 2^2 + \cdots + n\,2^n = \sum_{k=1}^{n} k\,2^k \ ,$$

$$v_n = 3^1 + 3^2 + \cdots + 3^n = \sum_{k=1}^{n} 3^k \ .$$

Donner un équivalent simple de u_n et de v_n, puis comparer leur croissance. (Indication pour u_n : l'écrire comme $2 \times S'_n(2)$, avec $S_n(x) = \sum_{k=1}^{n} x^k$.)

3 Notion d'algorithme

3.1 Variables et assertions

3.1.1 Variables

Un algorithme utilise des données mémorisées dans des **variables**. Il s'agit d'une part de variables **locales** à l'algorithme, d'autre part de variables passées en **paramètre** à la procédure. Les valeurs de ces variables peuvent être modifiées par la procédure.

(Techniquement parlant, les paramètres ne sont pas des *variables*, mais des *valeurs* transmises (souvent résultat de l'évaluation d'une expression comme par exemple lors d'un appel d'une fonction PGCD(3,6) ou arctan(PI/2)), qui sont copiés dans des variables locales. Dans certains langages de programmation, ces variables ne sont pas modifiables. Même lorsqu'ils le sont, une éventuelle modification par la procédure n'a pas d'effet "à l'extérieur" (en particulier, après l'execution) de la procédure. Pour que la procédure puisse *modifier* la valeur d'une variable "*extérieure*", il faut qu'elle connaisse son *adresse*, afin d'y modifier le *contenu*. Cependant, ce mécanisme (de référence par adresse) peut se faire, selon le langage de programmation, de façon plus ou moins automatique et donc "transparente", c-à-d. invisible, au niveau du "code" du programme.)

A chaque appel de l'algorithme, la place pour les variables locales et les paramètres est crée sur une même **pile** (principe LIFO : last in, first out). La pile redescend à son niveau initial après l'execution de l'algorithme.

Dans un **langage typé**, chaque variable ne peut recevoir qu'un **type de donné** bien précis (nombre entier ou rationnel de taille et/ou précision maximale bien définie, chaîne de caractère de longueur fixe ou variable,...), qui doit être **déclaré** en même temps que la variable, avant son utilisation.

Lorsque l'algorithme décrit une fonction, il faut ajouter une zone de mémoire qui reçoit le **résultat** calculé et "*renvoyé*". Théoriquement, il serait préférable que l'effet d'un algorithme soit le "renvoi" des résultats, et non des *effets secondaires* sur des variables "extérieures", plus difficiles à contrôler et à étudier. Dans la pratique, cela peut nécessiter la création d'une copie de l'ensemble des données qui serait inutilement coûteuse (on pensera au tri d'une grande base de données), et on accepte alors que l'algorithme fasse subir un traitement à ces données "in situ".

3.1.2 Assertions

Pendant l'execution de l'algorithme, afin de contrôler ce que fait l'algorithme, on est conduite à énoncer des conditions logiques reliant les variables, telles que $j = i + 1$ ou $i < j$. Ces conditions sont souvent appelées "assertions". Elles portent non sur les variables elles-mêmes, mais sur les valeurs contenues dans ces variables à un moment donné de l'executeion de l'algorithme. Elles peuvent être vraies ou fausses à un moment donné de cette execution. Plus généralement, on définit des conditions reliant différents états du système (informatique) à des *moments différents*, par exemple concernant la valeur d'une variable *après*

execution d'une instruction, en fonction de la valeur d'une autre variable *avant* cette execution.

Pendant l'execution d'un algorithme, les points (temporels) choisis pour l'évaluation de ces assertions sont ceux qui séparent les instructions, voir plus bas.

La définition et l'étude de l'algorithme conduit à énoncer des assertions vraies, caractérisant le but (travail à effectuer) ou le déroulement de ce traitement, en vue d'une *preuve* ou de l'évaluation de la *complexité* de l'algorithme.

3.2 Instructions, pré- et post-conditions

Une **instruction** modifie le système d'une façon définie à l'avance, pourvu qu'*avant* son execution une certaine assertion (dépendante de l'instruction), appelée la **pré-condition** de cette instruction, soit vérifiée. Dans ce cas, *après* son execution, la **post-condition** de l'instruction sera vérifiée. Celle-ci décrit donc le "but" de l'instruction (au sens large, car il peut être utile de préciser dans la post-condition aussi certains effets "secondaires" pas forcément souhaitées, tels que la modification éventuelle d'une variable ayant une valeur indéfinie après...). A défaut, on sous-entend que toute variable non mentionnée dans la post-condition ne sera pas modifiée ; ce qui est important pour contrôler le bon déroulement de l'algorithme par la suite.

On cherchera a distinguer les valeurs *avant* et *après* l'execution d'une instruction en introduisant un minimum de notation. A défaut d'autres précisions, s'il s'agit d'une variable x, on désignera par x sa valeur avant, et par x' sa valeur après l'execution.

L'**affectation** est l'instruction fondamentale. Elle est de la forme

```
variable <- expression .
```

L'execution de l'affectation se déroule en deux temps :

1. évaluation de l'expression,
2. transfert de la valeur dans la variable.

La *pré-condition* consiste par conséquent également en deux parties :

1. d'une part, que l'expression puisse être évaluée : les variables nécessaires convenablement définies, les fonctions accessibles et leurs arguments dans le domaine convenable,...
2. d'autre part, que la valeur résultant de l'expression soit compatible avec le type de la variable (surtout s'il s'agit d'un langage typé), ou qu'elle puisse être (implicitement) convertie en un tel type.

La *post-condition* est que la variable contient la *valeur évaluée de l'expression*, éventuellement convertie (arrondi, ...) selon le type de la variable.

Exemple 3.1 *Pour l'affectation* `x <- x + 1` *, la post-condition est* $x' = x+1$*, (*x *étant la valeur avant, et* x' *la valeur après l'execution). La pré-condition est*

que la valeur de x soit bien définie, et d'un type numérique (ou qu'elle puisse être convertie en un tel type), et enfin que la valeur de $x+1$ puisse encore être stockée dans la variable x (pas de dépassement des valeurs admissibles...). Dans la pratique peuvent s'ajouter d'autres aspects, par exemple l'equation $x' = x+1$ doit être considérée modulo la précision des données. Par exemple, si x est d'un type pouvant stocker des rationnels avec une précision de 10 chiffres décimaux, alors pour $x = 10^{12}$, la valeur x' coïncide avec celle de x, dans la limite de la précision considérée.

Remarque 3.2 *La simple évaluation d'une expression par le système informatique est une tâche bien plus complexe qu'on ne puisse penser. Elle comprend l'analyse lexique et syntactique, faisant intervenir la* ***précédence*** *(« priorité ») et* ***associativité*** *des opérateurs (par exemple, que $1+2\times 3$ nécessite d'abord l'évaluation de 2×3, mais $1-2-3$ nécessite en premier l'évaluation de $(1-2)$, que $-4!$ signifie $-(4!)$ et non $(-4)!$,... ; on parle de l'****arborescence de l'expression****, sa valeur étant à la racine, les embranchements étant les opérateurs (ou fonctions), auxquels sont attachés les arguments par des branches, se terminant en les feuilles qui sont les constantes). Il y a aussi certaines règles propres à chaque langage : par exemple, pour évaluer l'expression booléenne « $P(x)$ et $Q(x)$ », si $P(x)$ est faux, on n'est pas obligé d'évaluer $Q(x)$; lors d'une telle* ***évaluation paresseuse****, « $(x \neq 0$ et $1/x < \varepsilon)$ » ne donne pas d'erreur pour $x = 0$. Il n'est donc pas forcément une pré-condition que toutes les sous-expressions soient bien définies à l'avance. Or, l'évaluation de $Q(x)$ pouvant avoir des effet secondaires, il faut bien savoir quel type d'évaluation a lieu. Pour la même raison, il faut être sûr de l'*ordre dans lequel sont évalués les arguments *à une fonction, lorsqu'on écrit par exemple $f(g(x), h(y))$.*

Convention. Par prudence, on prendra recours à un stockage de résultats intermédiaires dans des variables temporaires, à des alternatives (si ... alors ...) développées, et à l'ajout de parenthèses, partout où il pourrait y avoir la moindre ambiguité. □

3.3 Structures de contrôle

La conception d'un algorithme présuppose que le système soit déjà capable d'executer des instructions plus élémentaires que l'algorithme lui-même.

Une **structure de contrôle** a pour but de contrôler le déroulement dans le temps des instructions I_1, I_2, I_3,... de l'algorithme. Elle utilise pour ce faire souvent des **conditions logiques**, dites *expressions booléennes*, notées C ci-après.

3.3.1 Séquence

Une **séquence** I_1, I_2, I_3,... I_n d'instructions signifie que ces instructions sont à executer dans l'ordre donné. Parfois, par exemple pour augmenter la lisibilité, on regroupe une telle séquence dans un bloc :

```
Faire
  I1 ; I2 ; ... ; In
Fin faire.
```

Au point d'execution suivant I_k, on a bien sûr la post-condition de I_k, et donc après la séquence la post-condition de I_n. La pré-condition de la séquence est la conjonction de la pré-condition de I_1 et de ce qu'il faut ajouter à la post-condition de chaque I_k pour obtenir la pré-condition de I_{k+1} ($1 < k < n$).

3.3.2 Alternative (si ... alors ...)

L'**alternative (simple)** est de la forme

```
Si ( C ) [alors]  I1  Fin si.
```

(Le mot-clé "`alors`" est parfois omis.)

La post-condition est : post-condition de I_1 ou non C ; autrement dit : $C \Rightarrow$ (post-condition de I_1).

L'**alternative équilibrée** est de la forme

```
Si ( C ) [alors]  I1  sinon  I2  Fin si.
```

La post-condition est : (C et post-condition de I_1) ou (non C et post-condition de I_2).

Remarque 3.3 *Dans les deux cas précédents, C signifie la valeur **avant** l'execution, qui n'est pas nécessairement égale à la valeur C' après.*

Remarque 3.4 *On trouve aussi des alternatives multiples, de la forme*

- *`Si (` C_1 `) alors` I_1 `sinon si (` C_2 `) alors` I_2 `... sinon` I_n `Fin si`*
- *`Selon (` expr `) cas` valeur$_1$ `:` I_1 `; cas` valeur$_2$ `:` I_2 `;... Fin selon`*

mais nous allons réaliser ces construction à l'aide d'alternatives simples ou équilibrées emboîtées.

3.3.3 Boucle (tant que)

Le seul type de **boucle** que nous allons utiliser dans la suite du cours est de la forme

```
Tant que ( C ) [faire]  I1  Fin tant que.
```

(Le mot-clé "`faire`" est parfois omis ; s'il est présent, on peut aussi remplacer le "`Fin tant que`" par "`Fin faire`".)

La *post-condition* est : non C, également appelée *condition d'arrêt*, alors que C est dite *condition de poursuite (de la boucle)*.

Il existent d'autres types de boucle,

- `Faire` I_1 `tant que (` C `)`
- `Faire` I_1 `jusqu'à ce que (` C `)`

– `Pour` *variable* `allant de` *initial* `à` *limite* `[par pas de` *incrément*`]` `faire` I_1 `Fin pour`
– `Pour (` *initialisation* `;` *cond.de poursuite* `;` *incrémentation* `) faire` I_1 `Fin pour`
– `Pour tout` *variable* `element de` *expression* `faire` I_1 `Fin pour tout`

Moyennant des alternatives et/ou autres instructions supplémentaires (ou une duplication des instructions I_1 avant le début de la boucle), toutes ces variations peuvent se réaliser en termes d'une boucle `tant que`. Cette dernière étant la plus élémentaire et simple à analyser, nous n'allons considérer que ce type de boucle.

Remarque 3.5 *Les alternatives et les boucles sont appelées* ***instructions complexes****, par opposition à des instructions élémentaires* I_1, I_2,...

Remarque 3.6 *Dans chacune des instructions complexes présentées dans ce chapitre, les symboles* I_1 *et* I_2 *peuvent représenter chacun toute une séquence ou un bloc d'instructions élémentaires ou complexes, ce qui peut conduire par exemple à la notion de* ***boucles emboîtées*** *etc..*

3.3.4 Autres structures de contrôle

Il existent d'autres structures de contrôle, permettant par exemple

1. la terminaison de l'algorithme avec renvoi du résultat, ailleurs qu'à la fin de l'algorithme ("`return`"),
2. l'interruption et sortie d'une boucle (ou autre structure complexe) à un endroit quelconque, indépendamment de la condition d'arrêt ("`break`"), ou le "saut" à l'étape suivante ("`continue`"),
3. la continuation de l'algorithme à un point d'execution quelconque ("`go to`").

Nous n'allons pas envisager l'utilisation de telles instructions (que l'on peut toujours éviter en modifiant l'algorithme), car elle rend en général plus difficile l'étude des algorithmes.

3.4 Algorithme

3.4.1 Définition

Pour définir un algorithme il est essentiel de préciser :

1. les données sur lequel il porte : paramêtres transmis à la procédure et/ou éventuellement modifiés par la fonction,
2. la pré-condition qui définit les valeurs autorisées des données pour que l'algorithme s'exécute correctement,
3. la post-conction, qui correpond à ce que l'algorithme doit faire,
4. l'algorithme proprement dit, qui consiste en une séquence d'instructions généralement complexes, c-à-d. comportant des structures de contrôle.

3.4.2 Exemple : Euclide (calcul du PGCD)

L'algorithme suivant porte sur les données : m et n.

Pré-condition : m et n sont deux entiers positifs.

Post-condition : la variable $PGCD$ contient le PGCD de m et de n. (On utilise a, b et r comme variables locales dont on ne précise pas la valeur finale.)

L'algorithme :

```
a <- max( m, n ) ; b <- min( m, n )
Tant que ( b > 0 )
  r <- Reste_division( a, b )
  a <- b
  b <- r
Fin tant que
PGCD <- a
```

(On aurait pu renoncer à la première ligne, et remplacer a, b par m, n.)

3.4.3 Structure fréquente d'algorithmes

Les algorithmes étudiés dans ce cours seront souvent divisés en trois parties :

1. une initialisation, (dans l'exemple ci-dessus, ce qui précède le "Tant que")
2. une boucle, (ci-dessus, les lignes du "tant que" jusqu'au "fin tant que")
3. une finalisation (ci dessus, l'affectation de la variable PGCD).

(En principe, tout algorithme peut se mettre sous cette forme.)

Lorsque l'algorithme ne fait pas appel à lui-même, on l'appelle "**itératif**", chaque étape de l'execution (de la suite des instructions à l'intérieur) de la boucle étant appelée une **itération**.

Algorithmes récursifs Les algorithmes faisant appel à eux-mêmes sont dits **récursifs**. On pourrait réaliser tout algorithme sous forme récursive, sans utiliser de boucle. Cependant, un algorithme récursif est généralement plus difficile à étudier et plus coûteux à executer (complexité spatiale, voir le chapitre suivant) que la version itérative.

Exemple 3.7 *Le calcul du PGCD se réalise de manière récursive comme suit :*

```
PGCD( m, n ):  Si ( m = 0 ) alors PGCD <- n
               sinon PGCD <- PGCD( Reste_division( n,m ), m ).
```

Remarque 3.8 *On pourrait aussi remplacer* `Reste_division( n,m )` *par la valeur absolue* $|n-m|$. *En effet, si m et n sont strictement positifs, on diminue strictement au moins l'un des paramètres, en passant de (m, n) à $(|n - m|, m)$ (exercice : démontrer ceci par disjonction de cas). Si $n = 0$, on calcule $PGCD(m, m)$, puis $PGCD(0, m)$, ce qui donne m. Ainsi l'algorithme s'arrête forcément au bout d'un nombre fini de récursions. Il suffit donc de vérifier que $PGCD(m, n) = PGCD(|n - m|, m)$, pour être sûr que le résultat est correct.*

4 Preuve d'un algorithme

4.1 Correction partielle

Dfinition 4.1 *Un algorithme est* ***partiellement correct*** *ssi, lorsqu'il s'arrête, il a fait ce qu'il doit faire.*

Exemple 4.2 *L'algorithme suivant cherche la racine carée d'un nombre a :*

```
n <- 0
tant que ( n*n != a ) /* != signifie : non egal */
  n <- random(0,a) /* nombre entier aleatoire entre 0 et a */
fin tant que
RacineCarree <- n
```

Cet algorithme est partiellement correct, car il ne s'arrête que si la condition $n^2 \neq a$ est fausse, donc si $n^2 = a$ est vérifée ; le résultat $RacineCarree = n$ est donc égal à la racine carée de a.

Un algorithme partiellement correct peut ne pas être satisfaisant, car on n'a pas la garantie qu'il s'arrêtera. Dans l'exemple précédent, il ne s'arrêtera jamais, si a n'est pas le carrée d'un entier.

4.2 Correction (terminaison)

Dfinition 4.3 *Un algorithme est* ***correct*** *ssi, il est partiellement correct* ***et*** *il s'arrête nécessairement, lorsque les données initiales vérifient sa pré-condition ; on parle alors de* ***terminaison****.*

Dans ce cas, on est sûr qu'il fera ce qu'il doit faire. Cependant, là encore, on n'a imposé aucune contrainte sur le nombre d'itérations possibles, ni sur la place de mêmoire disponible pour stocker des variables. Lors de l'implémentation sur un ordinateur ayant une vitesse de calcul et une mémoire finie, cela pourra durer arbitrairement longtemps, ou devenir impossible manque de mémoire. D'où l'intérêt de l'étude de la *complexité* de l'algorithme, au chapitre suivant.

Exercice 4.2.1 *Démontrer que l'algorithme suivant est correct.*
Il doit chercher la partie entière de la racine carrée d'un réel a positif.

```
n <- 0
Tant que ( n*n <= a ) /* inferieur ou egal */
  n <- n + 1
Fin tant que
RacineCarree <- n-1
```

4.3 Invariant de boucle

La preuve d'un algorithme est facilitée par la mise en évidence d'une assertion dite invariant de boucle.

Dfinition 4.4 *Une assertion est un invariant d'une boucle à un point d'execution α donné, si lorsqu'elle est vraie en α, alors au passage suivant, s'il a lieu, au même point α de la boucle, elle sera encore vraie.*

Le point α d'execution choisi est souvent celui qui précède la première instruction à l'intérieur de la boucle. Lorsque l'évaluation de la condition C de continuation de la boucle ne modifie pas l'état du système (ce que nous allons supposer dans la suite et nous efforcer de respecter lors de la conception d'algorithmes), alors le système est au point α dans le même état qu'au point β, situé après la dernière instruction de la boucle, lors du passage précédent.

4.4 Preuve de correction partielle

Ayant trouvé un invariant I qui représente le travail progressivement fait par la boucle pour atteindre le but de l'algorithme, on procède en 3 étapes :

1. On prouve, grâce à la pré-condition, qu'à l'issue de l'initialisation, I est vérifiée.
2. On suppose I vérifié en α et on prouve que I est vérifiée en β. C'est donc un invariant de boucle.
3. On prouve que la condition d'arrêt jointe à l'invariant I entraîne la post-condition, après la finalisation.

4.5 Preuve de la terminaison

Elle s'appuie sur
- la mise en évidence d'une expression qui progresse de façon strictement monotone à chaque tour de boucle, par exemple par incrémentation ou décrémentation fixe,
- le fait que le franchissement d'un certain seuil de cette expression provoque la condition d'arrêt de la boucle.

Exemple 4.5 *Dans beaucoup de cas simples, la condition d'arrêt est de la forme $A > B$, et la différence $A - B$ augmente d'une unité à chaque execution de la boucle.*

4.6 Exemple de preuve : tri par insertion

On suppose donné un tableau t à n éléments ($n \geq 1$) tous comparables entre eux. On souhaite le trier, c-à-d. réarranger les éléments en ordre croissant.

On considère ici la méthode du **tri par insertion**, qui consiste à mettre, pour i allant de 2 à n, l'élément $t[i]$ à "sa" place parmi les premiers i éléments, en décalant d'une case vers la droite ceux parmi $t[1..i-1]$ supérieurs à $t[i]$. Soit :

```
i <- 2
Tant que ( i <= n )
  Inserer_a_sa_place_parmi_les_precedents_l_element( i, t )
  i <- i+1
Fin tant que
```

1. L'assertion « $t[1..i-1]$ est trié » est l'invariant de boucle : elle est vérifée initialement, pour $i = 2$, et pourvu que l'insertion se fasse correctement, on peut affirmer que sous cette pré-condition, après chaque étape, la zone $t[1..i]$ sera triée, donc que l'invariant sera vérifé pour $i' = i + 1$, c-à-d. à nouveau en début de l'étape suivante.
2. L'algorithme ne termine que si la condition d'arrêt $i > n$ est vérifiée ; on a alors $i = n + 1$ (l'incrémentation de i n'ayant lieu que si $i \leq n$), et donc que $t[1..n]$ est trié : l'algorithme est partiellement correct.
3. L'incrémentation de i tant que $i \leq n$ implique qu'à terme la condition d'arrêt $i > n$ sera vérifiée : l'algorithme est donc correct.

Pour être complet, détaillons le (sous-)algorithme

```
Inserer_a_sa_place_parmi_les_precedents_l_element( i, t ):
  x <- t[i]    /* sauvegarde pour decalage de t[i-1] vers t[i] */
  j <- i - 1
  Tant que ( j > 0 et t[j] > x )
    t[j+1] <- t[j] /* decaler l'element courant d'une place */
    j <- j - 1     /* (dont l'occupant a deja ete copie avant). */
  Fin tant que
  t[j+1] <- x     /* t[j] <= x ou j = 0 (tout est deja decale) */
```

Remarque 4.6 *On aurait pu*

- *vérifier au début si $t[i-1] \leq t[i]$, auquel cas rien n'est à faire ; ou encore à la fin, si $j + 1$ est vraiment différent de i,*
- *utiliser i comme variable locale à la place de j (on ne peut alors plus simplement copier ce « code » dans la boucle principale du tri par insertion)*
- *éviter la sauvegarde $x \leftarrow t[i]$ et l'insertion finale de x, en faisant à la place des permutations successives des éléments $t[j]$ et $t[j+1]$; or, une telle permutation est en général réalisée sous la forme $x \leftarrow t[j]$, $t[j] \leftarrow t[j+1]$, $t[j+1] \leftarrow x$, et nécessite alors trois fois plus d'opérations élémentaires !*
- *faire d'abord la recherche de la "bonne place", et ensuite séparément le décalage puis l'insertion ; ici il nous a paru plus simple de faire ces deux opérations progressivement et simultanément.*

5 Notion de complexité

5.1 Introduction

Dans la pratique, le fait qu'on algorithme soit correct ne suffit pas pour être sûr qu'il fournira le résultat souhaité. En effet, son execution peut exiger trop de temps ou de mémoire que nous n'avons à notre disposition. Même un algorithme qui paraît simple à l'homme, peut être trop complexe pour la machine du point de vue de l'execution. Ceci est surtout le cas pour des algorithmes récursifs.

Dans le monde informatique contemporain, certains types de données ont atteint des grandeurs impressionants :

1. nombre de points sur une page A4 à 360ppp : 12 000 000
   (ppp = points par pouce, en anglais : dpi = dots per inch)
2. nombre de pages web recensées par google en 2005 : 9 000 000 000
3. nombre de caractères que peut stocker un disque dur : 50 000 000 000
   (un caractère correspondant à un octet, unité de mémoire pouvant prendre 256 valeurs différentes)

Ainsi il est essentiel de pouvoir juger l'efficacité d'algorithmes lorsque les données ont une taille importante ; pour cela il faut disposer d'outils permettant de mesurer leur complexité.

5.2 Complexité temporelle et spatiale

5.2.1 Complexité temporelle

Dfinition 5.1 *La **complexité temporelle** correspond au temps que prend l'algorithme pour s'executer, en fonction d'un entier n représentant la taille de la donnée qui lui a été transmise.*

Certaines instructions ont des coûts temporels négligeables, telles que, par exemple, l'incrémentation ou comparaison d'entiers d'une certaine taille maximale caractéristique du microprocesseur (de l'ordre de $2^{31} \approx 10^9$ ou $2^{63} \approx 10^{20}$ actuellement). D'autres, comparant, transférant ou manipulant des objets complexes (chaînes de caractères, tableaux, graphes...) sont beaucoup plus couteux.

On fait généralement l'approximation qu'un **traitement élémentaire** d'un **objet caractéristique** de l'algorithme ait un temps d'execution moyen qui puisse servir d'**unité de temps**. Seulement exceptionellement, il sera nécessaire distinguer deux ou plusieurs différents traitements possibles pour ces objets : stockage, comparaison, opération(s) arithmetique(s),...

Pour évaluer la complexité d'un algorithme, il suffit dans la plupart des cas de ne considérer que les traitements le plus coûteux, exécutés de nombreuses fois, soit à l'intérieur d'une boucle, soit de manière récursive. par les algorithmes.. La complexité temporelle devient alors le nombre de fois que sont executés ces traitements.

5.2.2 Complexité spatiale

Dfinition 5.2 *La **complexité spatiale** mesure l'espace mémoire nécessaire à l'algorithme pour s'executer, en fonction d'un entier n représentant la taille de la donnée qui lui a été transmise.*

Dans le cas d'un algorithme non récursif, la taille des variables locales reste généralement limitée. Par contre, dans la cas d'un algorithme récursif, le même espace pour les variables locales doit être réservé sur une **pile** à chaque appel récursif. Le niveau maximal d'appels récursifs « emboîtés »déterminera alors la complexité spatiale de l'algorithme.

S'il s'agit d'un traitement de données très volumineuses en mémoire, elles ne seront généralement pas stockés comme variable locale sur la pile : leur place est alors allouée ailleurs dans la mémoire de l'ordinateur (voire dans une base de données externe), et seul un lien (pointeur, nom de fichier ou autre identifiant) vers cette mémoire est stocké comme variable locale et transmis lors d'appels récursifs ou à d'autres fonctions. Dans ce cas, la complexité spatiale doit aussi tenir compte de cette mémoire. On distinguera alors la mémoire (maximale) temporairement nécessaire pendant l'execution de l'algorithme (et libérée après), et la mémoire occupée de façon persistante, après l'execution, par le résultat du traitement.

Dans ce cours, nous n'allons pas étudier des cas où il y a une telle allocation de mémoire en dehors des variables locales.

5.3 Complexité moyenne, au pire et meilleur des cas

La complexité ne dépend que de l'entier n qui représente la taille de la donnée traitée par l'algorithme. Elle varie selon cette donnée de taille n entre une valeur maximale, dite **complexité au pire des cas**, et une valeur minimale, dite **complexité au meilleur des cas**.

Si les données de taille n prennent des valeurs $(v_i)_{i\in I}$ avec des probabilités $(p_i)_{i\in I}$, la **complexité en moyenne** est donnée par

$$C_m(n) = \sum_{i\in I} p_i\, C(v_i)\ ,$$

où $C(v_i)$ est la complexité pour la donnée v_i. En particulier, on aura $p_i = f_i/\sum f_k$, si f_i est la fréquence d'apparition de v_i, et la somme ce ces fréquences est finie, et encore plus simple, $p_i = 1/N$, s'il n'y a qu'un nombre fini N de valeurs possible, ayant tous la même fréquence. $C_m(n)$ est donc la moyenne des complexités possibles, pondérées par leur probabilité, pour des données de taille n.

5.4 Classes de complexité

Il n'est pas nécessaire ni utile de connaître la fonction de complexité très précisément. La capacité et la puissance des ordinateurs variant d'un facteur 1 à 100, mais guère plus (sauf cas exceptionnels, tels que ordinateurs de poche par rapport à des supercalculateurs), un facteur constant dans la complexité d'un algorithme ne modifie pas le critère d'arrêt en temps raisonnablent limité ou non. Ainsi il suffit, pour évaluer la complexité, de déterminer sa classe d'équivalence grossière :

Classe	$\Theta(C)$	Commentaire
logarithmique	$\log(n)$	très rapide
linéaire	n	très rapide
quasilinéaire	$n \log n$	rapide
quadratique	n^2	moyennement rapide
polynômiale	n^α $(\alpha > 2)$	faisable sur ordinateur pour n pas trop grand
exponentielle	$e^{\alpha\, n} = k^n$ $(\alpha > 0,\ k > 1)$	infaisable même pour des valeurs
factorielle ou plus	$n!^\alpha,\ e^{f(n)}$ $(f' \to \infty)$	pas très grandes de n

Exercice 5.4.1 *On suppose que la fonction de complexité $c(.)$ d'un algorithme vérifie $c(1) = 1$ et pour tout $n \in \mathbb{N}$: $c(n) \leq 2\, c(n \div 2) + 1$ (où $n \div 2$ signifie la partie entière de $\frac{n}{2}$).*

1. *Majorer $c(100)$, en appliquant l'inégalité de manière récursive.*
2. *Majorer $c(2^p)$, en cherchant une sur-suite $(v_0, v_1, ...)$ de $u_p = c(2^p)$.*
3. *Montrer que cette majoration est valable pour tout entier n au lieu de 2^p. [Solution : $c(n) \leq 2\, n - 1$, complexité linéaire.]*

6 Exemples de calculs de complexité

6.1 Recherche linéaire dans un tableau

Pour chercher une valeur x dans un tableau T indexé de 1 à n, en l'absence d'hypothèse supplémentaire, il faut comparer $T[i]$ à x pour tout i s'incrémentant de 1 à n, tant que $T[i] \neq x$.

Pour évaluer la complexité temporelle de cet algorithme, on ne comptera que les comparaisons de x aux $T[i]$, supposées prépondérantes devant les autres instructions (incrémentation de i,...).

On a alors :

– complexité au meilleur des cas = 1

– complexité au pire des cas = n

En supposant que chaque case du tableau ne peut prendre que p valeurs, il y a p^n tableaux possibles, dont exactement

$(p-1)^{k-1}.p^{n-k}$ tableaux où x apparaît pour la première fois dans la case $T[k]$, et $(p-1)^n$ tableaux où x n'apparaît pas. On montre alors que

– complexité en moyenne $\sim p$, lorsque $n \to \infty$.

Remarque 6.1 *Cet exemple est un peu académique dans la mesure où, si p est raisonnablement petit, il suffirait de coder les differents éléments par un entier, auquel cas la comparaison ne nécessite pas plus de temps que la manipulation de la variable de boucle. Dans le cas contraire (p très grand, par exemple les valeurs possibles de nombres décimaux dans une précision donnée, typiquement* $p \approx 2^{32} \approx 10^{10}$ *ou* $p \approx 2^{64} \approx 10^{20}$*), la complexité au pire des cas* $(= n)$ *sera toujours largement inférieure à la limite de la complexité "moyenne"* $(= p)$*, car on n'aura pratiquement jamais un tableau à* $n \geq p$ *éléments (la mémoire d'un disque dur entier étant limité au stockage de l'ordre de* $n \approx 10^9$ *tels nombres).*

6.2 Algorithme dichotomique

On considère ici le cas fréquent d'un algorithme de la forme

```
/* Debut */
Initialisation

/* Boucle */
Tant que ( C )
   /* point alpha */
   Instructions
   /* point beta */
Fin tant que
/* Fin de boucle */

Finalisation
/* Fin */
```

Supposons qu'on puisse définir une expression N, en fonction des variables, telle que

- lorsque N passe de la valeur N_p, au point α (avant le début des instructions dans la boucle), à la valeur N_{p+1}, au point β (après les instructions dans la boucle), alors on a $N_{p+1} \leq N_p/2 + c$ (avec un certain c fixé, par exemple $c = 1$),
- la condition $N_p < s$ (où $s = \sup\{\, s \mid s = s/2 + c \,\}$) est équivalent à l'arrêt (donc à la négation de la condition C de poursuite) de la boucle
- la valeur initiale $N_0 = n$ de N représente la taille des données

et que

- le temps d'execution des Instructions de la boucle ne varie pas considérablement et sera donc être pris comme unité de temps,
- le temps pour l'initialisation et la finalisation pourra être négligé.

Alors, $N_{p+1} - S \leq (N_p - S)/2$, où $S = S/2 + c$, d'où $N_p - S \leq (n - S)/2^p$, et lorsque ceci sera inférieur à c pour la première fois, la condition d'arrêt sera atteinte en début de la boucle suivante, et la valeur de p est un majorant de la complexité temporelle de l'algorithme. Or, $(n-S)/2^p \geq 1$ donne une complexité $\leq p \leq \log_2(n - S) = O(\log_2(n))$. On a donc montré

$$\text{complexité} = O(\log_2(n)) \ .$$

Si de plus on a

- lorsque N passe de N_p au point α à N_{p+1} au point β, alors $N_{P+1} \geq (N_p/2) - d$ (avec par exemple $d = 1$)

alors la relation inverse « complexité domine $\log_2(n)$ »est obtenue de même manière, d'où l'équivalence grossière

$$\text{complexité} = \Theta(\log_2(n)) \ .$$

Ce type d'algorithme tient son nom de l'exemple type qui sont les algorithmes basées sur la méthode consistant à diviser un travail en deux moitiés égales et à s'appliquer recusivement à chacune des moitiés. Un exemple sera discuté en détail dans le paragraphe suivant ; voir aussi la méthode "diviser pour régner" à la fin du cours.

6.3 Recherche dichotomique dans un tableau trié

On souhaite à nouveau chercher une valeur x dans un tableau $T[i..j]$, mais on suppose maintenant ce tableau trié. La recherche dichotomique consiste alors à le diviser (si $i < j$) en deux moitiés $T[i..k-1]$ et $T[k+1..j]$, et à continuer la recherche dans la moitié choisie après comparaison de x avec $T[k]$. On montre alors que l'algorithme est dichotomique au sens précédent (avec $n = j - i$).

Pour obtenir un résultat intéressant au cas où la valeur de x n'est pas présente dans $T[1..n]$, précisons que l'algorithme doit trouver la *place* du dernier élément **égal ou inférieur** à x (où 0 s'il n'y en a pas) : ainsi on sait que

- x est présent dans le tableau $\iff$ $place > 0$ et $t[place] = x$
- pour insérer x dans le tableau (qui doit rester trié), il faut nécessairement décaler tous les éléments d'indice $k > place$ d'une case vers la droite,
- on a toujours $place \in [[0, n]]$.

D'où l'algorithme suivant :

```
Si ( x < t[1] ) alors place <- 0
sinon:
   Si ( x >= t[n] ) alors place <- n
   sinon:
      i <- 1 ; j <- n
      Tant que ( i+1 < j )  /* invariant: t[i] <= x < t[j] */
         k <- (i+j) div 2
         Si ( t[k] > x ) alors j <- k
         sinon i <- k
         Fin si      /* invariant: t[i] <= x < t[j] */
      Fin tant que
      place <- i   /* post-condition:                    */
   Fin si          /*       (place > 0 => t[place] <= x)  */
Fin si             /*    et (place < n => t[place+1] > x) */
```

On voit qu'à chaque étape, $i < k < j$, donc l'intervalle est strictement réduit, d'où la terminaison. A l'arrondi près, sa taille est divisé par 2 à chaque étape, d'où :

Complexité, au pire des cas, de la recherche dichotomique dans un tableau de taille n :

$$C(n) \sim \log_2(n) \qquad (n \to \infty) \ .$$

6.4 Exponentiation entière

L'algorithme suivant donne une méthode efficace pour calculer une puissance entière m^n d'un objet m pour lequel une multiplication est définie (nombre décimal, matrice, polynôme...). On évalue le coût en terme de multiplications d'objets du type de m, sachant qu'une telle multiplication (de matrice, polynôme, nombre décimal) pourra nécessiter un nombre plus ou moins important de multiplications et/ou autres opérations plus élémentaires.

L'algorithme dichotomique présenté ci-dessous sera d'une complexité de $\log_2 n$. Il s'agit en effet d'utiliser l'écriture de n en base 2,

$$n = a_0 + a_1\, 2 + a_2\, 2^2 + a_3\, 2^3 + \cdots + a_q\, 2^q$$

d'où la formule

$$m^n = m^{a_0}\ \left(m^2\right)^{a_1} \left(m^{2^2}\right)^{a_2} \left(m^{2^3}\right)^{a_3} \cdots \left(m^{2^q}\right)^{a_q}$$

où les a_i sont les restes successifs de la division de n par 2 (soit $a_i \in \{0, 1\}$) et les puissances m^{2^i} seront calculés comme carré de $m^{2^{i-1}}$. Ceci donne l'algorithme suivant :

```
p <- 1
Tant que ( n > 0 )
   Si ( n impair )
      p <- p * m
   Fin si
   m <- m * m /* double la puissance de m_initial */
   n <- n div 2 /* div. entiere */
Fin tant que
puissance <- p
```

L'invariant de boucle est : $p \cdot m^n = m_0^{n_0}$, où les valeurs indexées par 0 correspondent aux valeurs initiales, c-à-d. aux valeurs de la formule mathématique ci-dessus.

Dmonstration. L'invariant est vérifié initialement, lorsque $p = 1$, $m = m_0$, $n = n_0$. Montrons que lorsqu'il est vérifié en début de boucle, au point α juste avant ou après "tant que", alors il est vérifié en fin de boucle, au point β juste avant "fin tant que". On note avec une prime les grandeurs au point β, et sans prime les valeurs au point α. On a alors

$$p' = p\, m^a,\ m' = m^2,\ n = 2\, n' + a,\ \text{où } a \in \{0, 1\}\ ,$$

$$p\, m^n = p\, m^{2n'+a} = p\, m^a\, (m^2)^{n'} = p'\, (m')^{n'} \quad (CQFD).$$

Lorsque l'algorithme termine, on a $n = 0$ (car n ne prend que des valeurs entières positives), donc $p = (m_0)^{n_0}$; ainsi l'algorithme est partiellement correct.

D'autre part, tant que n est strictement positif, il sera divisé par 2, ce qui diminue strictement sa valeur. Ceci implique que l'algorithme terminera au bout d'un nombre fini d'itérations de la boucle. L'algorithme est donc correct.

Le nombre d'itérations avant que la division de n par 2 ne donne zéro est égal au nombre q ci-dessus, qui est la plus haute puissance de 2 inférieure ou égale à n, ou encore la partie entière (et donc équivalent, pour $n \to \infty$) de $\log_2(n)$. La complexité temporelle de l'algorithme est donc celle de $\log_2(n)$.

6.5 Tri par ségmentation

Commençons par un exercice pour motiver la suite :

Exercice 6.5.1 *Etudier la complexité temporelle du tri par insertion.*
Indication : vérifier qu'à l'étape i, il y a entre 1 et $i-1$ opérations (comparaison + décalage) nécessaires ; en déduire qu'en moyenne, la complexité est quadratique ; seulement au meilleur des cas (nombre négligeable de décalages, c-à-d. tableau déjà presque trié) elle est linéaire.

Exercice 6.5.2 *En supposant que seul les comparaisons ont un coût non-négligeable (justifié si l'on utilise une liste dynamique au lieu du tableau, permettant une insertion sans décalage), étudier la complexité du tri par insertion* ***amélioré****, utilisant une recherche dichotomique pour trouver la place où il faut insérer l'élément $t[i]$.*
Comparer au tri par insertion simple, pour $n = 100\,000$.
Indication : on passe à $c_{amélioré}(n) = \sum \log_2(i-1) \sim n \log_2(n)$. Le rapport avec $c_{simple}(n) \sim \frac{1}{2} n^2$ est $5 \times 10^4 / \log_2(10^5) \approx 3010$.

6.5.1 Ségmentation d'un tableau

Dans un premier temps, on souhaite diviser une zone $t[g..d]$ d'un tableau t en deux parties $t[g..i-1]$ et $t[i..d]$ telles que tous les éléments de la première zone soient inférieurs à tous les éléments de la deuxième zone. Cela nécessitera en général des permutations de certains éléments. En voici l'algorithme :

```
/* Initialisation. (On suppose g < d.) */
v <- t[d] ; i <- g ; j <- d

/* Boucle 0 : Invariant I0 : (I1 et I2), voir ci-dessous */
Tant que ( i < j )

   /* Boucle 1 : Invariant I1 : pour tout x de t[g..i-1], x <= v */
   Tant que ( i <= j et t[i] <= v )
      i <- i+1  /* on cherche le premier element > v */
   Fin tant que /* fin de boucle 1 */

   /* Boucle 2 : Invariant I2 : pour tout x de t[j+1..d], x >= v */
   Tant que ( i <= j et t[j] >= v )
      j <- j-1  /* on cherche le dernier element < v */
   Fin tant que /* fin de boucle 2 */

   Si ( i < j ) /* ce cas se produit, si
                   l'element i dans [g..j-1] est superieur a v,
               ET l'element j dans [i+1..d] est inferieur a v */
      permuter t[i] et t[j]
      /* ainsi maintenant t[i] < v < t[j]
```

```
        et la boucle 1 suivante augmentera i d'au moins 1 */
   Fin si
Fin tant que /* fin de boucle 0 */
/* les zones t[g..i-1] et t[i..d] verifient alors la propriete souhaitee
   (N.B.: on peut en effet avoir j=i-1 (c-a-d. j<i) ou alors j=i.) */
```

6.5.2 Tri par segmentation

Ayant ainsi segmenté le tableau en deux parties d'éléments tous inférieurs, dans la zone gauche, à tous ceux de la zone droite, on applique (de manière récursive) le même algorithme aux deux moitiés, jusqu'à ce que la zone à trier ne contient plus qu'un seul élément. Le but est alors atteint, c-à-d. le tableau est trié dans sa globalité.

Cette procédure est un cas particulier de la méthode générale de "diviser pour régner" (voir ci-dessous). Si la coupure se fait en moyenne au milieu du tableau, le temps T nécessaire pour trier un tableau de taille n vérifie l'équation $T(n) = 2\,T(n/2) + f(n)$, où $f(n)$ est le temps nécessaire pour la segmentation.

6.6 Méthode "diviser pour régner"

Si on divise un traitement de n données dans un temps $f(n)$ en un nombre a de tâches portant sur n/b données (avec généralement $a \geq b$, mais souvent $a = b$), alors le temps d'execution de l'algorithme vérifie

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

L'intérêt de ceci est conséquence du résultat suivant :

Theorme 6.2 *Soit $T(n) = a\,T(n/b) + f(n)$, avec $a, b \geq 1$. Dans ce cas,*

1. *si $f(n) = O(n^{\log_b a - d})$ avec $d > 0$, alors $T(n) = \Theta(n^{\log_b a})$,*
2. *si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$,*
3. *si $f(n) = \Omega(n^{\log_b a + d})$ et $a\,f(n/b) \leq c f(b)$, avec $d > 0$, $c < 1$, alors $T(n) = \Theta(f(n))$.*

Dans le dernier cas, f est la partie la plus coûteuse, on ne peut donc espérer mieux que l'équivalence (grossière) donnée (il est plutôt agréablement surprenant que les executions récursives de f ne modifient pas l'echelle) ; dans les deux autres cas, lorsque f est dominé par une échelle en puissance $n^{\log_b a}$ (avec un exposant en général proche de 1, ce cas (linéaire) étant atteint pour $a = b$), alors T est essentiellement de cette même complexité, considérée comme rapide.

Exemple 6.3 *Pour le tri par segmentation, en supposant encore que la coupure se fasse (en moyenne) au milieu du tableau, on a $a = b = 2$ et $f(n) = \Theta(n)$ (on compare v à chacun des éléments ; au pire des cas, il faut en plus faire la permutation à chaque fois), d'où $T(n) = O(n \log n)$.*

FIN