Algorithmes : efficacité, analyse et ordre de complexité

(Résumé du chapitre 1 du manuel)

Table des matières

1 Algorithmes : efficacité, analyse et ordre de complexité

1.1 Qu'est-ce qu'un algorithme?

Définition de la notion d'algorithme

– Définitions du terme "algorithme"

- Selon le Petit Robert :

  > Ensemble des règles opératoires propres à un calcul. Calcul, enchaînement des actions nécessaires à l'accomplissement d'une tâche.

- Selon le Grand dictionnaire terminologique (`www.granddictionnaire.com`) :

  > Ensemble des règles opératoires qui permettent la résolution d'un problème par l'application d'un nombre fini d'opérations de calcul à exécuter en séquence.

- Selon `dicofr.com` :

  > Un jeu de règles ou de procédures bien défini qu'il faut suivre pour obtenir la solution d'un problème dans un nombre fini d'étapes. [...] Un algorithme peut être simple ou compliqué. Cependant un algorithme doit obtenir une solution en un nombre fini d'étapes.

– Une caractéristique importante d'un algorithme est que son exécution *doit se terminer*, après l'exécution d'un nombre fini d'étapes. Ce n'est pas le cas de tous les programmes.

Algorithme et programme

– Un algorithme doit être traduit en un langage de programmation pour produire un *programme* compilable et exécutable. Par contre, ceci ne signifie pas nécessairement qu'un programme définisse un algorithme :

- Un système est dit *réactif* lorsqu'il maintient une interaction constante avec son environnement et lorsque son comportement est dirigé par les événements (*event-driven*). Les événements sont liés soit à des stimuli internes ou externes, soit à des contraintes liées à l'écoulement du temps. L'objectif d'un système réactif n'est donc pas de produire un ultime résultat final mais bien *d'interagir* avec son environnement.
- Un système est dit *fonctionnel* ou *transformationel* (en anglais, *functional* ou *transformational*) lorsqu'il produit à partir de données d'entrée un ensemble de sorties puis *termine* son exécution. De tels systèmes sont aussi dit *input-output driven.*

Les systèmes de traitement en lots (*batch*) sont des systèmes fonctionnels. Par contre, des réseaux de télécommunication, des systèmes d'exploitation, des systèmes de contrôle de procédés, des systèmes embarqués, des interfaces personnes–machines, etc., sont plutôt des systèmes réactifs.

De ces définitions et exemples, on peut comprendre qu'un programme réactif *ne devrait jamais terminer son exécution* et, donc, *ne réalise* pas un algorithme, Évidemment, ceci n'empêche pas qu'un tel programme, à l'intérieur des diverses tâches qu'il doive exécuter, puisse utiliser et mettre en oeuvre divers algorithmes.

Algorithme : paradigme de programmation et modèle architectural

– Un algorithme doit, ultimement, être *traduit* dans un langage de programmation, de façon à pouvoir résoudre de façon effective des problèmes à l'aide de programmes et systèmes informatiques.

Un programme est un ensemble de composants écrits dans un ou plusieurs langages de programmation et exécutables sur un type particulier de machine $\Rightarrow$ un programme comporte de nombreux détails, n'est pas aussi *abstrait* qu'un algorithme.

Une caractéristique typique d'un algorithme est la suivante : il s'agit d'une solution exprimée de façon relativement *abstraite*, de façon *indépendante* d'un langage ou d'un compilateur particulier, sans référence à une machine spécifique.

Toutefois, comme on le verra au cours de la session, lorsqu'on élargit notre éventail de paradigmes de conception d'algorithmes, donc lorsqu'on connaît de nombreuses façons d'aborder la résolution de problèmes algorithmiques, on s'aperçoit alors qu'un algorithme *ne peut pas* être indépendant ...

- du choix d'un style (d'un paradigme) de programmation
- du choix d'un modèle architectural (vers lequel l'algorithme sera, ultimement, traduit et exécuté)

En d'autres mots, pour paraphraser l'expression qui dit que "lorsque le marteau est le seul outil qu'on connaît, alors on voit des clous partout", on peut dire que si le seul modèle architectural qu'on connaît est celui de von Neumann, alors on voit des algorithmes séquentiels et impératifs partout ...

Dans la première partie du cours, on supposera l'utilisation :

- Du paradigme de programmation procédurale ou objet
- D'une architecture séquentielle classique (modèle de *von Neumann*)

Toutefois, quelques exemples utilisant le paradigme de programmation fonctionnel seront aussi présentés.

Dans la deuxième partie du cours, on abordera ensuite la présentation de divers paradigmes de programmation parallèle.

Notations pour les algorithmes

– Tout comme il existe de nombreux langage de programmation, il existe de nombreuses "notations" qui peuvent être utilisées pour exprimer des algorithmes. Différentes notations seront utilisées dans le cadre du cours.

- Pseudocode lié au langage C++ tel qu'utilisé dans le manuel de référence : Algorithme 1 (Algorithme 1.2, p. 7 du manuel).

```
number sum( int n, const number S[] )
{
 index i;
 number result;

 result = 0;
 for( i = 1; i <= n; i++ )
  result = result + S[i];
 return result;
}
```

Algorithme 1: Somme des éléments d'un tableau : notation du manuel

- Pseudocode françisé : Algorithme 2.

```
PROCEDURE sum( n: Nat, S: Entier[] ): Entier
DEBUT
   result <- 0;
   POUR i <- 1 A n FAIRE
      result <- result + S[i];
   FIN
   RETOURNER( result )
FIN
```

Algorithme 2: Somme des éléments d'un tableau : pseudocode des notes de cours

- Langage MPD, un langage de programmation (donc compilable et exécutable) de haut niveau (basé sur les langages Pascal *et* C) : Algorithme 3.

```
procedure sum( int n, int S[*] ) returns int result
{
  result = 0;
  for [i = 1 to n] {
    result += S[i];
  }
}
```

Algorithme 3: Somme des éléments d'un tableau : notation MPD

De ces courts exemples, on peut constater, par l'exemple en notation MPD, qu'une notation peut être compilable et exécutable *tout en étant relativement succincte.*

Quelques explications concernant l'exemple MPD (Algorithme 3) :

- Par défaut, un argument est transmis *par valeur* (y compris un tableau).
- Le symbole "`*`", lorsqu'utilisé pour spécifier les bornes d'un paramètre formel tableau, indique qu'un tableau de taille arbitraire peut être reçu et traité (ici, l'argument `n` nous donne la taille de ce tableau).
- Une fonction (par opposition à une procédure) est définie à l'aide d'une clause `returns` indiquant le type du résultat et une variable qui contiendra la valeur produite par la fonction.

- Une variable d'itération, donc locale à une boucle `for` (par ex., `i`), n'a pas besoin d'être explicitement déclarée.
- Il n'est pas nécessaire d'utiliser une instruction `return` pour retourner le résultat — en fait, l'instruction `return`, contrairement à celle de C, ne prend aucun argument. Le résultat produit par la fonction est simplement la valeur finale de la variable introduite par la clause `returns`.

1.2 De l'importance de développer des algorithmes efficaces

– Depuis de nombreuses années, les machines sont de plus en plus rapides et puissantes, avec de plus en plus de mémoire... mais les problèmes qu'on aborde sont souvent de plus en plus complexes, donc de plus en plus gourmands en temps et en ressources

⇒ le choix d'un algorithme efficace reste, et restera toujours, important!

1.2.1 Fouille séquentielle vs. fouille binaire

```
procedure binsearch( int n,
                     int S[*],
                     keytype x,
                     res int location )
# PRECONDITION
#   n >= 0,
#   ALL( 1 <= i < n :: S[i] <= S[i+1] )
# POSTCONDITION
#   SOME( 1 <= i <= n :: S[i] = x )
#     => (1 <= location <= n) & S[location] = x,
#   ALL ( 1 <= i <= n :: S[i] ~= x )
#     => location = 0
{
  int l = 1;
  int h = n;
  location = 0;
  while( l <= h & location == 0 ) {
    int mid = (l + h) / 2;
    if ( x == S[mid] ) {
      location = mid;
    } else if ( x < S[mid] ) {
      h = mid-1;
    } else { # x > S[mid]
      l = mid+1;
    }
  }
}
```

Algorithme 4: Algorithme pour fouille binaire en notation MPD

Exemples :

- Un algorithme pour fouille binaire est présenté aux pages 9–10 du manuel (Algorithme 1.5). Un algorithme identique, mais exprimé dans la notation MPD, est présenté à l'Algorithme 4.

  Quelques remarques concernant cet exemple :

- Les variables `low` et `high` de l'algorithme du manuel ont été renommées `l` et `h` : en MPD, `low` et `high` sont des fonctions pré-définies du langage permettant d'obtenir la valeur la plus petite ou la plus grande d'un type (par ex., `high(int)` retourne 2147483647 sur un Pentium III).
- Un argument formel déclaré de mode `res` indique un argument utilisé pour retourner un `rés`ultat (*copy-out*, i.e., `OUT` en Ada). Le mode `var` indique un argument utilisé pour recevoir une valeur et retourner un résultat (*copy-in/copy-out*, i.e., `IN OUT` en Ada).
- Dans plusieurs des exemples que nous étudierons, nous tenterons de spécifier de façon explicite (et relativement *formelle*) les pré/post-conditions décrivant le comportement des principales procédures et fonctions. Ces pré/post-conditions seront spécifiées à l'aide de commentaires (caractère "`#`" en MPD). La notation utilisée, qui est basée sur le langage formel de spécification `Spec` (logique des prédicats du premier ordre) est expliquée brièvement à la fin du présent document (p. 21)).

- Nombre de comparaisons effectuées (dans le pire cas) par une fouille séquentielle en comparaison avec une fouille binaire :

Taille du tableau	Fouille séquentielle	Fouille binaire
128	128	8
1 024	1 024	11
1 048 576	1 048 576	21
4 294 967 296	4 294 967 296	33

1.2.2 Nombres de Fibonacci

- Algorithme récursif pour le calcul du $n$ième nombre de Fibonacci : Algorithme 5.

  Nombre de termes dans l'arbre récursif de calcul de `fib(n)` $= \Omega(2^n)$ (p. 15 du manuel)

```
procedure fib( int n ) returns int r
# PRECONDITION
#   n >= 0
{
   if (n <= 1) {
     r = n;
   } else {
     r = fib(n-1) + fib(n-2);
   }
}
```

Algorithme 5: Calcul des nombres de Fibonacci : version récursive

- Algorithme itératif pour le calcul du $n$ième nombre de Fibonacci : Algorithme 6.
- Temps d'exécution des versions récursive et itérative pour différentes valeurs de `n` si on suppose que chaque terme peut être calculé en 1 ns (10^{-9} sec) :

`n`	Temps pour `fib` (récursif)	Temps pour `fib2` (itératif)
40	1 048 000 ns	41 ns
80	18 min	81 ns
160	38 000 000 années	161 ns

1.3 Analyse des algorithmes

Analyse d'un algorithme = déterminer, de façon relativement *abstraite* (c'est-à-dire, indépendante d'un langage ou d'une machine), son efficacité (en temps et/ou en espace).

```
procedure fib2( int n ) returns int r
# PRECONDITION
#   n >= 0
{
   int f[0:n];

   f[0] = 0;
   if (n > 0) {
     f[1] = 1;
     for [i = 2 to n] {
       f[i] = f[i-1] + f[i-2];
     }
   }
   r = f[n];
}
```

Algorithme 6: Calcul des nombres de Fibonacci : version itérative

L'analyse d'algorithmes permet de comparer divers algorithmes entre eux, donc permet de choisir celui qui est le plus *efficace* (en temps et/ou en espace, selon le cas).

1.3.1 Analyse de la complexité temporelle

– Analyse de la complexité temporelle d'un algorithme $\neq$ analyse exacte du temps d'exécution du programme associé... parce que cela dépend trop du langage, du compilateur, de la machine utilisée.

Analyse de la complexité temporelle = déterminer, de façon indépendante du langage et de la machine, le nombre d'opérations *élémentaires* (opérations de base) nécessaires pour résoudre un problème *en fonction de sa taille.*

– Le choix des opérations élémentaires à analyser/compter *et* du paramètre qui détermine la taille du problème à résoudre ... dépendent du problème à traiter.

Exemples :

- Tri d'un tableau d'entiers arbitraires :
  - Opérations élémentaires = comparaisons entre deux éléments (parce que le nombre total d'opérations sera, *grosso modo*, proportionnel au nombre de comparaisons)
  - Taille = nombre d'éléments du tableau
- Tri d'un tableau d'éléments compris dans un intervalle limité de valeurs :
  - Opération élémentaire = Ajout d'un élément dans une liste et fusion des listes
  - Taille = nombre d'éléments
- Multiplication de matrices :
  - Opérations élémentaire = multiplications et additions
  - Taille = taille des matrices (nombre de lignes/colonnes)

– Selon Brassard et Bratley (1996, p. 54, ma traduction), "une opération élémentaire en est une dont le temps d'exécution peut être borné par un constante qui dépend seulement de la mise en oeuvre — la machine, le langage de programmation, etc. Donc cette constante *ne dépend pas* de la taille du problème ou de tout autres paramètres de l'instance du problème".

– Certains auteurs (Brassard et Bratley, 1996, ma traduction) parlent de l'utilisation d'une *opération barométrique* pour analyser un algorithme : "Une opération barométrique est une opération qui est exécutée au moins aussi souvent que n'importe quelle autre instruction de l'algorithme." Comme on cherche uniquement à obtenir l'*ordre de complexité* (l'ordre de grandeur) de l'algorithme, il n'y a donc aucun problème à ce qu'une opération non barométrique soit exécutée un nombre constant de fois plus souvent qu'une opération barométrique. En d'autres mots, l'utilisation d'une opération barométrique permet d'éviter de manipuler de façon explicite des constantes de multiplicité dans la définition de l'ordre de complexité. Pour plus de détails sur l'utilisation d'une opération barométrique pour analyser un algorithme, voir plus bas (Annexe C).

– Dans la plupart des algorithmes, mais pas tous, le nombre d'opérations effectuées ne dépend pas uniquement de la taille des données mais aussi des données elles-mêmes. Par exemple :

- Somme des éléments d'un tableau de taille n : nombre d'additions $= n - 1$ peu importe le contenu du tableau (à moins qu'on fasse un traitement spécial pour l'élément 0, auquel cas chaque addition devra être précédée d'une comparaison) $\Rightarrow n$ opérations
- Fouille séquentielle :
  - L'élément cherché est le premier au début de la liste $\Rightarrow$ une (1) comparaison
  - L'élément cherché est le dernier à la fin de la liste $\Rightarrow n$ comparaisons

Soit n la taille d'un problème et $T(n)$ le nombre exact d'opérations élémentaires effectuées par l'algorithme pour un problème de taille n

Différents types d'analyse de complexité temporelle :

- Complexité de tous les cas (*every case*) $= T(n)$ pour un n quelconque, si une telle fonction existe
- Complexité du pire cas $= W(n) =$ nombre d'opérations dans le pire des cas (c'est-à-dire, nombre maximum d'opérations requis)
- Complexité du meilleur cas $= B(n) =$ dual de pire cas
- Complexité moyenne $= A(n) =$ nombre moyen (espéré) d'opérations élémentaires requis pour un problème de taille n

Voir Parenthèse 1 pour plus de détails sur ces différents types d'analyse de complexité.
Exemples :

- Complexité de tous les cas pour la multiplication de matrices (pas de "pire" cas)
- Complexité du pire cas pour la fouille séquentielle
- Complexité du cas moyen pour la fouille séquentielle (pp. 21–22 du manuel)

Les types d'analyse les plus couramment utilisées :

- Pire cas : généralement la plus facile à déterminer
- Temps moyen : plus complexe à déterminer pcq. on doit associer une distribution de probabiblités aux différentes données possibles à l'entrée

La plus couramment utilisée dans ce cours :

- Pire cas (plus simple au niveau mathématique)

Il est possible de définir de façon un peu plus rigoureuse, mais quand même relativement informelle, les différents types d'analyse de complexité temporelle.

Soit A un algorithme. Notons par I l'ensemble, possiblement infini, de toutes les entrées possibles pour cet algorithme. Notons par I_n l'ensemble, fini, des entrées de taille n.

Pour une entrée $x \in I$, notons par $t(x)$ le nombre d'opérations barométriques (ou d'opérations élémentaires, selon le cas) requis par l'algorithme A sur l'entrée x.

On aura alors les définitions suivantes des différents types d'analyse de complexité :

- Complexité du pire cas :
$$W(n) = \max_{x \in I_n} t(x)$$
- Complexité du meilleur cas :
$$B(n) = \min_{x \in I_n} t(x)$$
- Complexité dans tous les cas :
$$T(n) = W(n), \text{si } W(n) = B(n), \text{ sinon pas défini}$$
- Complexité moyenne :
$$A(n) = \frac{\sum_{x \in I_n} t(x)}{|I_n|}$$

De façon générale, il est souvent difficile de calculer $A(n)$ de façon théorique. Lorsqu'on le fait, ceci repose habituellement sur certaines hypothèses concernant la distribution possible des entrées, hypothèses qui peuvent ne pas correspondre aux données rencontrées en pratique.

Parenthèse 1: Définitions de $T(n), W(n), B(n)$ et $A(n)$

1.3.2 Application de la théorie

– Hypothèse de base = si on choisit correctement la ou les opérations élémentaires (ou les opérations barométriques), en ignorant les opérations secondaires (*overhead operations*), alors on obtient une bonne base pour comparer deux algorithmes

Mais... si la mise en oeuvre réelle de ces opérations élémentaires est très coûteuse, alors la comparaison peut ne pas être correcte, ne pas être fidèle

– Exemple :

- Algorithme A : $T(n) = n$
- Algorithme B : $T(n) = n^2$

Supposons que chaque opération élémentaire de A, pour un langage et une machine donnée, soit 1000 fois plus coûteuse à exécuter qu'une opération élémentaire de B :

- Algorithme A : $T(n) = 1000n$
- Algorithme B : $T(n) = n^2$

$\Rightarrow B$ sera plus rapide que A pour $n < 1000$

1.4 Ordre de complexité

1.4.1 Introduction intuitive à la notion d'ordre de complexité

L'ordre de complexité d'une fonction nous donne un ordre de grandeur de son *taux de croissance*. Plus l'ordre de complexité est élevé, plus le taux de croissance est rapide : Voir Figure 1.3 (p. 28 du manuel) et Table 1.4 (p. 29 du manuel).

– Différentes catégories de complexité (de la plus simple à la plus complexe) :

$\Theta(1)$	constant
$\Theta(\lg\ n)$	logarithmique
$\Theta(\lg^k\ n)\ (k > 1)$	polylogarithmique
$\Theta(n)$	linéaire
$\Theta(n\ \lg\ n)$	$n\ \lg\ n$
$\Theta(n^2)$	quadratique
$\Theta(n^3)$	cubique
$\Theta(n^k)\ (k > 3)$	polynomial
$\Theta(2^n)$	exponentiel
$\Theta(a^n)\ (a > 2)$	exponentiel
$\Theta(n!)$	factoriel

Note : $\lg^k\ n = (\lg\ n)^k$. Pour $a, b > 0$, on a $\lg^b\ n \in o(n^a)$.

1.4.2 Une définition rigoureuse de la notion d'ordre de complexité

Note : Dans ce qui suit, on suppose que les fonctions analysées (f, g, etc.) sont des fonctions non-négatives (dont les arguments et les résultats sont nuls ou positifs).

– **Définition** de la notation O (grand O) : Étant donné une fonction $f(n)$, $O(f(n))$ est l'ensemble des fonctions $g(n)$ telles qu'il existe une constante positive réelle c et un entier non négatif N tel que pour tout $n \geq N$

$$g(n) \leq c \times f(n)$$

– **Définition** (bis) de la notation O (grand O) :

$$g(n) \in O(f(n))$$

ssi

$$\exists c \in \mathcal{R}^+, N \in \mathcal{N}, \forall n \in \mathcal{N} \bullet n \geq N \Rightarrow g(n) \leq c \times f(n)$$

Informellement: La notation $O(f)$ décrit le comportement *asymptotique* d'une fonction, c'est-à-dire, pour des grandes valeurs. En d'autres mots, g est $O(f)$ lorsque, pour des valeurs suffisamment grandes de n ($n > N$), le rapport $\frac{g(n)}{f(n)}$ reste toujours borné par c.

Informellement: (bis) g est $O(f)$ ne signifie pas que g est plus petit que f. Cela dit plutôt que g n'est jamais plus que c fois plus grand que f.

Exemples :

- $n^2 + 10n \in O(n^2)$, parce que pour $n \geq 1$,
$$n^2 + 10n \leq n^2 + 10n^2 = 11n^2$$
donc avec $c = 11$ et $N = 1$.
- $n^2 + 10n \in O(n^2)$, parce que pour $n \geq 10$,
$$n^2 + 10n \leq n^2 + n^2 = 2n^2$$
donc avec $c = 2$ et $N = 10$.
- $n \in O(n^2)$, parce que, pour $n \geq 1$,
$$n \leq 1 \times n^2$$
donc avec $c = 1$ et $N = 1$.

– Illustration de la définition de $O(f)$. Pour simplifier, supposons que $f(n) \geq 0$ et $g(n) \geq 0$. Pour déterminer si $g(n)$ est $O(f(n))$ (g est dominée asymptotiquement par f), on doit trouver $N \in \mathcal{N}$ et $c \in \mathcal{R}$ tel que

$$n > N \Rightarrow g(n) \leq c \times f(n)$$

Intuitivement, on peut considérer deux cas distincts :

- Premier cas (figure 1) : À partir d'un certain point N ($n > N$), $g(n) \leq f(n)$. Dans ce cas, on a donc trouvé le c approprié ($c = 1$).

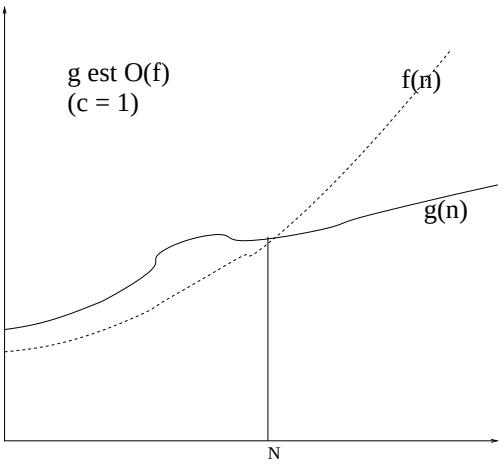

Figure 1: g est $O(f)$: g est plus petit ou égal à f à partir d'un certain point

- Deuxième cas (figure 2) : On a que $f(n)$ est plus petit que $g(n)$. Par contre, à partir d'un certain point N, $f(n)$ n'est jamais plus que c fois plus petit que $g(n)$. En d'autres mots, si on "gonfle" f par un facteur constant c, alors à partir d'un certain point N $(n > N)$ $g(n)$ est plus petit ou égal à $c \times f(n)$.

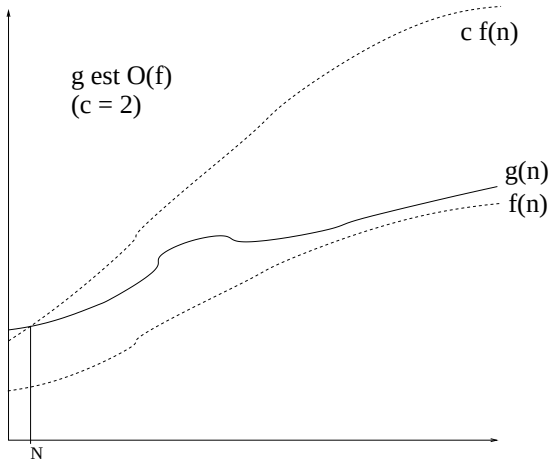

Figure 2: g est $O(f)$: g n'est pas plus petit que f mais le devient si on gonfle f un nombre constant de fois

– **Définition** de la notation Ω (grand oméga) : Étant donné une fonction $f(n)$, $\Omega(f(n))$ est l'ensemble des fonctions $g(n)$ telles qu'il existe une constante positive réelle c et un entier non négatif N tel que pour tout $n \geq N$

$$g(n) \geq c \times f(n)$$

En d'autres mots, $f(n)$ est une (*forme* de) borne inférieure asymptotique pour les fonctions dans $\Omega(f(n))$, alors qu'elle est une borne supérieure asymptotique pour les fonctions dans $O(f(n))$.

– **Définition** (bis) de la notation Ω (grand oméga) :

$$g(n) \in \Omega(f(n))$$

ssi

$$\exists c \in \mathcal{R}^+, N \in \mathcal{N}, \forall n \in \mathcal{N} \bullet n \geq N \Rightarrow g(n) \geq c \times f(n)$$

Si une fonction est à la fois dans $O(f(n))$ et dans $\Omega(f(n))$, alors on peut en conclure que son taux de croissance est *équivalent* à celui de $f(n)$.

– **Définition** de la notation Θ (grand théta) : Étant donné une fonction $f(n)$, $\Theta(f(n))$ est l'ensemble des fonctions $g(n)$ telles qu'il existe une constante positive réelle c_1, une constante positive réelle c_2 et un entier non négatif N tel que pour tout $n \geq N$

$$c_1 \times f(n) \leq g(n) \leq c_2 \times f(n)$$

En d'autres mots :

$$\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$$

– **Définition** de la notation o (petit o) : Étant donné une fonction $f(n)$, $o(f(n))$ est l'ensemble des fonctions $g(n)$ telles que *pour toute constante* positive réelle c, il existe un entier non négatif N tel que pour tout $n \geq N$

$$g(n) \leq c \times f(n)$$

En d'autres mots, $g(n)$ est dominée asymptotiquement de façon *stricte* par $f(n)$. Ainsi, on a la propriété suivante :

$$g(n) \in o(f(n)) \Rightarrow g(n) \in O(f(n)) - \Omega(f(n))$$

– La notation Θ sépare l'ensemble des fonctions de complexité en une collection de sous-ensembles disjoints (classes d'équivalence). Pour l'analyse des algorithmes, on utilise donc le représentant le plus simple de la classe, par ex., $\Theta(1), \Theta(\lg\, n), \Theta(n), \Theta(n\, \lg\, n), \Theta(n^2)$, etc.

1.5 Propriétés et simplifications de fonctions O, Θ et Ω

– Les propriétés suivantes de O pourraient être démontrées (mais nous ne le ferons pas en classe, peut-être en exercice `;)` :

- Supposons que $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$.
  Alors, $(f_1 + f_2)(n) \in O(g_1(n) + g_2(n))$.
- Supposons que $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$.
  Alors, $(f_1 + f_2)(n) \in O(max\{g_1(n), g_2(n)\})$.
- Supposons que $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$.
  Alors, $(f_1 \times f_2)(n) \in O(g_1(n) \times g_2(n))$.

Note importante : toutes ces propriétés restent valides si on remplace O par Ω ou Θ.
Ces propriétés sont utiles pour simplifier l'analyse d'algorithmes comme suit :

- Supposons que les temps d'exécution des opérations A et B soient respectivement $O(f(n))$ et $O(g(n))$. Alors le temps pour A suivi de B sera de $O(f(n) + g(n))$.

  Notons que dans le cas où, par exemple, f domine g de façon stricte ($g \in o(f)$), on peut alors en conclure que A suivi de B est de temps $O(f(n))$. De même, si $f \in \Theta(g)$, le temps sera alors simplement $O(f(n))$.
- Supposons que chaque exécution d'une boucle requiert un temps d'exécution $O(f(n))$ et que la boucle est exécutée $O(g(n))$ fois. Alors le temps sera $O(f(n) \times g(n))$.

Encore une fois, ces simplifications restent valides si on remplace O par Ω ou Θ.

– Autres propriétés :

1. $g(n) \in O(f(n))$ si et seulement si $f(n) \in \Omega(g(n))$
2. $g(n) \in \Theta(f(n))$ si et seulement si $f(n) \in \Theta(g(n))$
3. Si $b > 1$ and $a > 1$, alors
$$log_a\ n \in \Theta(log_b\ n)$$
(le logarithme utilisé n'a pas d'importance)
4. Si $b > a > 0$, alors
$$a^n \in o(b^n)$$
(toutes les fonctions exponentielles différentes sont dans des classes distinctes)
5. Pour tout $a > 0$,
$$a^n \in o(n!)$$
($n!$ est la pire des fonctions de complexité)
6. Soit les fonctions de complexité ordonnées comme suit, où $k > j > 2$ et $b > a > 1$:
$$\Theta(1)\ \Theta(\lg\ n)\ \Theta(n)\ \Theta(n\ \lg\ n)\ \Theta(n^2)\ \Theta(n^j)\ \Theta(n^k)\ \Theta(a^n)\ \Theta(b^n)\ \Theta(n!)$$
Si une fonction $g(n)$ est dans une catégorie à gauche de la catégorie contenant $f(n)$, alors $g(n) \in o(f(n))$.
(hiérarchie stricte)
7. Si $c \geq 0, d > 0, g(n) \in O(f(n))$, et $h(n) \in \Theta(f(n))$, alors
$$c \times g(n) + d \times h(n) \in \Theta(f(n))$$

De façon plus générale, la propriété 6 indique que toute fonction logarithmique est éventuellement meilleure que n'importe quelle fonction polynomiale, que toute fonction polynomiale est éventuellement meilleure que n'importe quelle fonction exponentielle, et que toute fonction exponentielle est meilleure que la fonction factorielle.

Les propriétés 6 et 7 peuvent être utilisées de façon répétitive pour simplifier des expressions et déterminer la catégorie la plus simple à laquelle appartient une fonction.

Exemple : On veut montrer que $5n + 3\ \lg\ n + 10\ n\ \lg\ n + 7n^2 \in \Theta(n^2)$:

- $7n^2 \in \Theta(n^2)$
- $10\ n\ \lg\ n + 7n^2 \in \Theta(n^2)$
- $3\ \lg\ n + 10\ n\ \lg\ n + 7n^2 \in \Theta(n^2)$
- $5n + 3\ \lg\ n + 10\ n\ \lg\ n + 7n^2 \in \Theta(n^2)$

Remarque sur les fonctions de complexité représentées par des polynômes

Cormen, Leiserson et Rivest (p. 31) présente la propriété suivante concernant les fonctions de complexité qui sont des polynômes.

Soit $p(n)$ un polynôme de degré d défini comme suit :

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

Lorsque $a_d > 0$, alors on a $p(n) \in \Theta(n^d)$ (et ce, donc, même si certains des autre coefficients sont négatifs).

Sauf dans les cas où on demandera explicitement d'appliquer les définitions de O, Ω ou Θ, on pourra donc utiliser cette propriété pour simplifier les expressions polynomiales.

Remarque sur la notation

De nombreux auteurs utilisent plutôt les notations suivantes :

- $f(n) = \Theta(n^2)$ plutôt que $f(n) \in \Theta(n^2)$.
- $f(n) = O(n^2)$ plutôt que $f(n) \in O(n^2)$.
- etc.

La notation ensembliste semble plus correcte en termes de définitions mathématiques et de la définition de "=". Toutefois, selon l'occasion, nous utiliserons l'une ou l'autre des notations, avec une préférence pour la notation ensembliste.

Remarques sur l'utilisation de O vs. Ω vs. Θ

La notation O décrit une borne supérieure. On peut donc l'utiliser pour obtenir une borne supérieure sur le temps d'exécution *dans le pire des cas*. Évidemment, ce faisant, on obtient aussi une borne supérieure pour des données quelconques.

Par contre, on peut aussi utiliser Θ pour décrire une borne, à la fois inférieure et supérieure, du temps d'exécution dans le pire des cas. Évidemment, cela ne signifie alors pas qu'une exécution de l'algorithme sur des données arbitraires sera bornée par Θ.

La notation Ω, quant à elle, décrit une borne inférieure. Habituellement, on l'utilise donc pour borner le temps d'exécution *dans le meilleur des cas*. Par contre, on peut aussi l'utiliser pour décrire une borne inférieure dans le pire cas.

Par exemple, soit l'algorithme 7, où la complexité asymptotique du temps d'exécution de chacune des parties est indiquée en commentaires.

```
PROCEDURE proc( n: Nat, Tab: Nat[] )
DEBUT
   POUR i <- 1 A n FAIRE
     SI test(i) ALORS
       C1      // Θ(1)
     SINON
       C2      // Θ(n)
     FIN
   FIN
FIN
```

Algorithme 7: Algorithme pour illustrer les différents types d'analyse

Toutes les affirmations suivantes concernant la complexité asymptotique du temps d'exécution de `proc` sont alors correctes :

- `proc` est $O(n^2)$.
- `proc` est $\Omega(n)$.
- Dans le pire des cas, `proc` est $\Theta(n^2)$. [1]
- Dans le meilleur des cas, `proc` est $\Theta(n)$.

Signalons que, par contre, si le temps pour le bout de code `C1` était plutôt $\Theta(n)$, on pourrait alors conclure plus simplement que `proc`, dans *tous les cas*, est $\Theta(n^2)$.

[1] Rappelons que cela implique évidemment que, dans le pire des cas, `proc` est $\Omega(n^2)$, ce qui est plus précis que simplement $\Omega(n)$.

A Analyse des diverses structures de contrôle

Règle générale, l'analyse d'un algorithme se fait de l'intérieur vers l'extérieur, c'est-à-dire, qu'on analyse les instructions individuelles, puis les structures qui les englobent, etc., jusqu'à obtenir le résultat pour l'algorithme dans son ensemble.

Dans ce qui suit, lorsqu'on indique qu'une opération est de temps t, on peut supposer dans un premier temps qu'on effectue une analyse détaillée du nombre d'opérations élémentaires exécutées par l'algorithme en fonction de la taille. Donc, un temps t signifie que l'algorithme va exécuter t opérations élémentaires (pour un problème d'une taille donnée). Si l'on effectue une analyse en se basant plutôt sur le choix d'une ou plusieurs opérations barométriques, le principe reste alors le même, mais l'analyse en est généralement simplifiée.

- Séquence :

  Soit A_1 et A_2 deux fragments d'algorithme, respectivement de temps t_1 et t_2. Le temps pour l'exécution de $A_1; A_2$ sera évidemment $t_1 + t_2$. Toutefois, si ces temps sont exprimés en termes de fonction de complexité, on pourra plus simplement conclure que le temps total sera $\Theta(max(t_1, t_2))$.

- Instruction `SI` :

  Soit l'instruction conditionnelle suivante, où le temps pour évaluer *condition* est t_0 alors que le temps pour exécuter A_i est t_i $(i = 1, 2)$:

  ```
  SI condition ALORS
    A1
  SINON
    A2
  FIN
  ```

  Le temps total sera alors $t_0 + \Theta(max(t_1, t_2))$.

- Boucle `POUR` :

  Soit la boucle suivante :

  ```
  POUR i <- 1 A n FAIRE
    A(i)
  FIN
  ```

  Notons par t_0 le temps, à chacune des itérations, pour gérer la variable d'itération `i`. Notons par $t(i)$ le temps pour l'exécution de l'itération `A(i)` — le temps $t(i)$, dans le cas général, peut évidemment dépendre de `i`. Le temps total sera donc le suivant :

  $$\sum_{i=1}^{n} [t_0 + t(i)]$$

  Notons toutefois que le temps t_0 est généralement un temps constant, donc d'ordre $\Theta(1)$. Le temps pour la boucle `POUR` sera donc toujours au moins $\Theta(n)$ $(\Theta(n) + \sum_{i=1}^{n} t(i))$.

  Évidemment, si $t(i)$ ne dépend pas de i, le temps total sera alors simplement $n \times [t_0 + t(i)]$.

- Boucle `TANTQUE` :

  Soit la boucle suivante :

```
TANTQUE condition FAIRE
  A
FIN
```

La première chose à vérifier d'une telle boucle est que, dans tous les cas possibles, son exécution se terminera.

Ensuite, il s'agit de déterminer le nombre de fois où la boucle sera exécutée. Malheureusement, il n'est pas toujours possible de déterminer à l'avance le nombre *exact* d'itérations. S'il s'agit d'une analyse du pire cas (resp. meilleur cas), on doit alors analyser la boucle pour obtenir une borne supérieure (resp. inférieure) pour le nombre d'itérations. On utilise ensuite cette information comme pour une boucle `POUR` (le temps de chaque itération peut ou non être constant).

- Procédures et fonctions récursives :

  On verra plus en détail, au chapitre 2 (Diviser-pour-régner), que l'analyse des algorithmes définis par des procédures ou fonctions récursives donne lieu à des équations de récurrence, donc une caractérisation *implicite*, équations qu'il faut alors résoudre pour trouver la solution *explicite*

Exemple d'analyse (boucle `TANTQUE`) pour l'algorithme 4 (p. 4) :

- La boucle se termine : pour montrer cette propriété, il faut vérifier que la condition `l <= h & location == 0` est assurée de devenir vraie. Notons tout d'abord que l'expression `h-l+1`, qui est toujours positive ou nulle,[2] nous donne le nombre de positions du tableau `S` où l'élément recherché `x` peut encore apparaître, *s'il est présent*. Notons par h et l la valeur de `h` et `l` *avant* la boucle, et par h' et l' la valeur *après* la boucle. À chaque itération, on a alors trois cas possibles :

  1. `x == S[mid]` : la boucle se termine immédiatement.
  2. `x < S[mid]` : on a $l = l'$, alors que $h' = \texttt{mid} - 1$, où $\texttt{mid} = (l+h)/2$. Il s'agit ici d'une division entière, donc plus précisément on a $(l+h)/2 = \lfloor \frac{l+h}{2} \rfloor$
     On a donc :

$$\begin{aligned} h' - l' + 1 &= \texttt{mid} - 1 - l' + 1 \\ &= (l+h)/2 - 1 - l + 1 \\ &= (l+h)/2 - l \\ &= \lfloor \frac{l+h}{2} \rfloor - l \\ &\leq \frac{l+h}{2} - l \\ &= \frac{l+h-2l}{2} \\ &= \frac{h-l}{2} \\ &< \frac{h-l+1}{2} \end{aligned}$$

     C'est-à-dire que $h' - l' + 1 < \frac{h-l+1}{2}$, donc l'expression décroit de façon stricte.
  3. `x > S[mid]` : la nouvelle valeur de `l` est `mid+1`, où `mid = (l+h)/2`. Par un raisonnement semblable au précédent, on peut montrer que la valeur de l'expression `h-l+1` décroit là aussi de façon stricte.

[2] Initialement, `l=1` et `n>=0` implique que `h>=0`. Cette valeur reste ensuite positive ou nulle à chaque itération, puisque `l <= mid <= h` et que soit le nouveau `l` est `mid+1` (si `mid=h` alors `h-l+1 = 0`), soit le nouveau `h` est `mid-1` (`l=mid` implique `h-l+1 = 0`).

Dans les deux cas où la boucle ne se termine pas de façon immédiate, la valeur de `h-l+1`, qui indique le nombre de positions possibles où `x` peut apparaître, si présent, *décroit* de façon stricte. La boucle est donc assurée de se terminer.

- À chaque itération, la valeur de l'expression `h-l+1` est divisée par 2. Initialement, `h-l+1 = n`, le nombre d'éléments du tableau. Le nombre total d'itérations, dans le pire cas, sera donc lg `n`.
- À chaque itération, le travail effectué est $\Theta(1)$. Le travail total effectué par la boucle `TANTQUE` est donc $\Theta(\lg\, n)$.

B Sommaire du pseudocode utilisé dans les notes de cours

Le pseudocode *français* qui est parfois utilisé dans les notes de cours est basé, un peu comme MPD, sur un mélange de concepts et structures inspirés de différents langages (Pascal, C, Ada, Java, MPD). Dans cette annexe, nous illustrons les principales caractéristique des cette notation. Soit l'algorithme 8.

```
CONSTANTE PAS_DEFINI = ...

PROCEDURE fib( n: Nat ): Nat
DEBUT
   A <- new Nat[0:n]
   POUR i <- 0 A n FAIRE
     A[i] <- PAS_DEFINI
   FIN
   RETOURNER fib'( n, A )
FIN

PROCEDURE fib'( n: Nat, A: ARRAY[*] OF Nat ): Nat
DEBUT
   SI A[n] != PAS_DEFINI ALORS
      RETOURNER A[n]
   FIN
   // Première rencontre de l'argument
   SI n == 0 || n == 1 ALORS
      r <- 1
   SINON
      r <- fib'(n-1, A) + fib'(n-2, A)
   FIN
   A[n] <- r
   RETOURNER r
FIN
```

Algorithme 8: Procédures `fib` et `fib'` illustrant le pseudocode français

Les principaux éléments du pseudo-code introduits par cet algorithme sont les suivants :

- Les sous-programmes sont introduits par des déclarations de `PROCEDURE`.

  Lorsqu'il s'agit d'une *fonction*, donc un sous-programme qui retourne un résultat pouvant être utilisé dans une expression, `PROCEDURE` et `FONCTION` sont l'un et l'autre employés pour déclarer la fonction. Toujours dans le cas d'une fonction, le type du résultat retourné par cette fonction est déclaré après les arguments formels (qui sont entre parenthèses), par exemple :

  ```
  PROCEDURE f( a1: T1, ..., an: Tn ): TypeResultat
  ```

  Dans ce cas, le résultat de la fonction est retourné explicitement à l'aide d'une instruction `RETOURNER` (comme en C/Java).

  Dans certains cas, un identificateur est parfois explicitement introduit pour dénoter la variable locale dénotant le résultat de la fonction, par exemple :

```
PROCEDURE f( a1: T1, ..., an: Tn ) r: TypeResultat
```

Dans ce cas, la valeur retournée par la fonction est simplement la valeur conservée dans la variable à la fin de l'exécution.

- Règle générale, les variables locales ne sont pas explicitement déclarées. Lorsqu'on veut spécifier explicitement, de façon dynamique, la taille d'un tableau, celui-ci peut être alloué à l'aide d'un `new` (comme en Java).
- Une instruction d'affectation est dénotée par "`<-`".
- Une boucle `POUR` correspond à une boucle avec un nombre prédéfini d'itérations (boucle `for`).
- Les opérateurs logiques les plus courants sont `==`, `!=` (non égal), `&&` (et), `||` (ou).
- Le symbole "`//`" indique le début d'un commentaire, qui se termine avec la fin de la ligne.
- Une autre structure de contrôle couramment utilisée est la boucle avec un nombre indéfini d'itérations :

```
TANTQUE condition FAIRE
  ...
FIN
```

Les principaux types de données utilisés sont les suivants (inspirés principalement du langage formel de spécification `Spec`) :

- Types de base : `Nat`, `Entier`, `Réel`, `Chaîne`, etc.
- Ensembles = collections *non-ordonnées* d'éléments de même type :

```
e1: set{Nat}
e1 = {20, 30}
add(20, e1) = e1
size(e1 U e1) = 2
e1 U {30..32} = {20, 30, 31, 32}
{x: Nat SUCH THAT x IN e1 :: x+1} = {21, 31}
```

- Séquences = suites ordonnées d'éléments de même type :

```
s1: sequence{Nat}
s1 = [20, 30, 30]
length(s1) = 3
add(10, s1) = [10, 20, 30, 30]
s1[1] = 20
head(s1) = 20
tail(s1) = [30, 30]
s1 || [30..32] = [20, 30, 30, 30, 31, 32]
length(s1 || s1) = 6
```

- Tuples = suites finies d'éléments de types possiblement différents, tel qu'illustré dans l'algorithme 9 où la fonction `trouverMinMax` retourne un couple (un 2-tuple) de `Nat`.

```
PROCEDURE trouverMinMax( s: sequence{Nat} ): (Nat, Nat)
# PRECONDITION
#   length(s) >= 1
DEBUT
  (leMin, leMax) <- (s[1], s[1])
  POUR i <- 2 A length(s) FAIRE
    SI s[i] < leMin ALORS leMin <- s[i] FIN
    SI s[i] > leMax ALORS leMax <- s[i] FIN
  FIN
  RETOURNER( leMin, leMax )
FIN
```

Algorithme 9: Algorithme de recherche du minimum et maximum d'une séquence d'éléments

- Dictionnaires = associations entre clés et définitions :

```
d1: map{Chaîne, Nat}
d1 = {["Colin", 16], ["Zoe", 10]; 0}
d1["Zoe"] = 10
d1["Mathieu"] = 0  //  Retourne la valeur par défaut pcq. clé non définie
domain(d1) = {"Colin", "Zoe"}  //  Ensemble des clés définies
range(d1) = {0, 10, 16}  //  Ensemble des définitions, incluant la valeur par défaut
```

Les pré/post-conditions ainsi que les assertions seront formulées à l'aide d'expressions de la logique des prédicats du premier ordre utilisant la notation du langage `Spec` :

- Logique propositionnelle :

```
  p => q  <=>  ~p | q
 ~(p & q) <=>  ~p | ~q          (Loi de De Morgan)
```

- Logique des prédicats, c'est-à-dire, utilisation de quantificateurs existentiels (il existe (au moins) un élément) et universels (tous les éléments) :
  - `SOME( i: Nat :: 2*i = i )` :
    il existe un nombre naturel `i` avec la propriété que `2*i = i`.
  - `~SOME( i: Nat SUCH THAT i ~= 0 :: 2*i = i )` :
    il n'existe aucun nombre naturel `i`, différent de `0`, ayant la propriété que `2*i = i`.
  - `ALL( i: Nat :: i < i+1 )` :
    pour n'importe quel naturel `i`, on a que `i < i+1`.
  - `ALL( i: Nat, j: Nat SUCH i < j :: i+1 < j+1 )` :
    pour n'importe quels nombres naturels `i` et `j` tel que `i` est inférieur à `j`, on a que `i+1` est inférieur à `j+1`.

C Trois façons d'analyser un algorithme

On peut dire qu'il y a trois (3) façons différentes d'analyser un algorithme, d'une façon tout d'abord très détaillée jusqu'à une façon plus directe :

1. On compte, *de façon exacte*, le nombre d'opérations élémentaires exécutées par l'algorithme. Une fois ce nombre obtenu, on trouve l'ordre de complexité associé en simplifiant à l'aide des propriétés de Θ.
2. On approxime, à l'aide d'un ordre de complexité (fonction Θ de base), chacune des parties de l'algorithme. On combine et simplifie ensuite ces différents ordres de grandeur à l'aide des propriétés (par ex., $O(f) + O(g) = O(max\{f, g\})$, etc.).
3. On identifie une *opération barométrique* appropriée et on calcule le nombre (exact) de fois où cette opération est exécutée, d'où l'on déduit ensuite l'ordre de complexité.

La différence importante entre une opération élémentaire est une opération barométrique est la suivante :

- Opération élémentaire = une opération dont le temps d'exécution peut être borné par un constante qui dépend seulement de la mise en oeuvre — la machine, le langage de programmation, etc.
- Opération barométrique = opération (élémentaire) qui est exécutée *au moins aussi souvent* que n'importe quelle autre instruction de l'algorithme. Lorsqu'on identifie cette opération barométrique, il est important de justifier brièvement pourquoi cette opération est effectivement exécutée au moins aussi souvent que les autres opérations.

Exemple pour l'algorithme suivant :

```
PROCEDURE sum( n: Nat, S: Entier[] ): Entier
DEBUT
   result <- 0;
   POUR i <- 1 A n FAIRE
      result <- result + S[i];
   FIN
   RETOURNER( result )
FIN
```

Utilisons comme opérations élémentaires les opérations de comparaisons, d'affectations ainsi que l'instruction de retour d'un résultat de fonction. Les différentes façons d'analyser cet algorithme conduiront alors aux résultats suivants :

1. Nombre exact d'opérations élémentaires :

$T(n)$	=	1	(initialisation de `result`)
		$+n(2+1)$	(n itérations, avec deux affectations (`i` et `result`) et une comparaison (test de la boucle) par itération)
		$+1$	(instruction `RETOURNER`)
	=	3n+2	
	$\in$	$\Theta(n)$	

2. Approximation de chacune des étapes (en termes d'opérations élémentaires) :

$$\begin{aligned} T(n) &= \Theta(1) + \Theta(n) * \Theta(1) + \Theta(1) \\ &= \Theta(1) + \Theta(n) + \Theta(1) \\ &= \Theta(n) \end{aligned}$$

3. Nombre d'exécutions d'une opération barométrique appropriée :

   Choisissons l'affectation à `result` comme opération barométrique : comme on s'intéresse au comportement asymptotique (donc pour de grands n), on peut voir que cette opération sera exécutée plus souvent que les instructions à l'extérieur de la boucle ou qu'elle sera exécutée aussi souvent que tout ce qui touche la manipulation de l'index (affectation ou test).

   Le nombre d'exécutions de cette opération sera alors le suivant :

$$\begin{aligned} T(n) &= n \\ &\in \Theta(n) \end{aligned}$$

D Remarque sur les fonctions de coût à plusieurs arguments

Lorsque les données d'un algorithme sont caractérisées par plusieurs arguments de tailles possiblement différentes, alors la fonction de complexité résultante peut très bien dépendre *de plusieurs arguments*. Par exemple, soit la procédure suivante :

```
PROCEDURE proc( s1: sequence{Nat}, s2: sequence{Nat} ) res: Nat
PRECONDITION
  n = length(s1)
  m = length(s2)
DEBUT
  Initialiser res
  POUR i <- 1 A n FAIRE
    POUR j <- 1 A m FAIRE
      Traiter s1[i] et s2[j] et mettre à jour res (en temps Θ(1))
    FIN
  FIN
FIN
```

S'il est important, pour cet algorithme, de distinguer entre la taille de `s1` et la taille de `s2` — par exemple, parce que ces séquences jouent des rôles différents, malgré leur type semblable —, alors la fonction donnant le temps d'exécution de l'algorithme devrait dépendre *de ces deux arguments*, d'où l'on concluerait que le temps d'exécution de cet algorithme est le suivant :

$$T(n, m) \in \Theta(n \times m)$$