Complexité des algorithmes [cx]
Algorithmique

Karine Zampieri, Stéphane Rivière, Béatrice Amerein-Soltner

Unisciel algoprog Version 21 mai 2018

Table des matières

Complexité des algorithmes

Mots-Clés Complexité des algorithmes ■
Requis Axiomatique impérative, Preuve et Notations asymptotiques, Récursivité des actions ■
Difficulté ● ● ○

Introduction

Deux paramètres essentiels servent à mesurer le coût d'un algorithme : le temps d'exécution et l'espace mémoire requis. En termes techniques, on parle respectivement de **complexité temporelle** et de **complexité spatiale**.

D'autres critères peuvent avoir de l'importance : le nombre d'échanges entre l'unité centrale et les disques magnétiques dans le cas de requêtes de Base de données, le trafic sur le réseau local auquel est connecté l'ordinateur, etc. Plus généralement, toute **ressource** mise à disposition de l'algorithme a un coût.

Ce module étudie principalement la complexité en temps d'exécution. Elle présente le contexte mathématique, définit les complexités en temps puis réalise l'étude de cas de la sous-séquence de somme maximale.

1 Théorie de la complexité

Complexité d'un algorithme

L'**efficacité** d'un algorithme se mesure par **le temps d'exécution** en fonction de la taille des données d'entrée et **la place mémoire** nécessaire à son exécution en fonction de la taille des données d'entrée. Ces deux fonctions sont appelées **complexité de l'algorithme**. La détermination de ces fonctions s'appelle l'**analyse de complexité**.

Dans la suite nous n'étudierons que le temps d'exécution. Pour mesurer le temps d'exécution comme fonction de la taille des données il faut se donner une unité de mesure. Pour simplifier le modèle on considère comme temps constant les opérations élémentaires exécutés par une machine (opérations arithmétiques, logiques, affectations, etc.). L'analyse consiste alors à exprimer le nombre d'opérations effectuées comme une fonction de la taille des données d'entrée.

Complexité asymptotique

Soient deux algorithmes $\mathcal{A}$ et $\mathcal{B}$ résolvant le même problème de complexité respectives $100n$ et n^2. Quel est le plus efficace ?

- Le rapport des complexités de `B` à $\mathcal{A}$ est égal à $n/100$. Donc :
  - Pour $n < 100$, $\mathcal{B}$ est plus efficace.
  - Pour $n = 100$, $\mathcal{A}$ et $\mathcal{B}$ ont la même efficacité.
  - Pour $n > 100$, $\mathcal{A}$ est plus efficace.
- Notons que plus n devient grand, plus $\mathcal{A}$ est efficace devant $\mathcal{B}$.

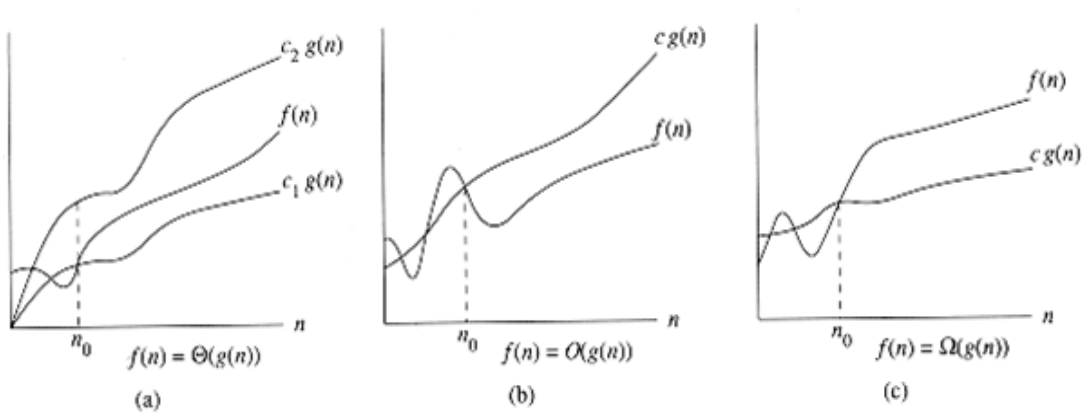
Si les tailles sont « petites », la plupart des algorithmes résolvant le même problème se valent. C'est le comportement de la complexité d'un algorithme quand la taille des données devient grande qui est important. On appelle ce comportement la **complexité asymptotique**.

Notations de Landau

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

Les **notations asymptotiques** sont définies comme suit :

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


$c_2\, g(n)$
$f(n)$
$c_1\, g(n)$
n
n_0
$f(n) = \Theta(g(n))$
(a)
$c\, g(n)$
$f(n)$
n
n_0
$f(n) = O(g(n))$
(b)
$f(n)$
$c\, g(n)$
n
n_0
$f(n) = \Omega(g(n))$
(c)

2 Complexités en temps d'un algorithme

Soient :

- D_n l'ensemble des données de taille n.
- $C(d)$ le coût d'exécution de l'algorithme sur la donnée d de taille n.

On définit plusieurs mesures pour caractériser le comportement d'un algorithme.

2.1 Complexité au meilleur

Complexité au meilleur
(Dite aussi complexité dans le **meilleur cas**) Est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à n :

$$T_{\min}(n) = \min_{d \in D_n} C(d)$$

Avantage
C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille n.

2.2 Complexité au pire

Complexité au pire
(Dite aussi complexité dans le **pire cas**, ou **cas le plus défavorable** (*worst-case* en anglais)) Est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à n :

$$T_{\max}(n) = \max_{d \in D_n} C(d)$$

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

Inconvénient
Cette complexité peut ne pas refléter le comportement « usuel » de l'algorithme, le pire cas pouvant ne se produire que très rarement, mais il n'est pas rare que le cas moyen soit aussi mauvais que le pire cas.

2.3 Complexité en moyenne

Complexité en moyenne
Est la moyenne des complexités de l'algorithme sur des jeux de données de taille n :

$$T_{\text{moy}}(n) = \sum\{\Pr(d) \cdot C(d),\ d \in D_n\}$$

où $\Pr(d)$ est la probabilité d'avoir la donnée d en entrée de l'algorithme.

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

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

En pratique
La complexité en moyenne est beaucoup plus difficile à déterminer que la complexité dans le pire cas, d'une part parce que l'analyse devient mathématiquement difficile, et d'autre part parce qu'il n'est pas toujours facile de déterminer un modèle de probabilités adéquat au problème.

Cas des configurations équiprobables
Si toutes les configurations des données de taille fixée n sont équiprobables, la complexité en moyenne s'exprime en fonction du nombre $|D_n|$ de données de taille n :

$$T_{\text{moy}}(n) = \frac{1}{|D_n|} \sum_{d \in D_n} C(d)$$

Remarque
Souvent on partitionne l'ensemble D_n des configurations de taille n selon leur coût, et on évalue la probabilité $\Pr(D_n^i)$ de chaque classe D_n^i des configurations de taille n de coût i. La complexité en moyenne devient alors :

$$T_{\text{moy}}(n) = \sum_{D_n^i \subseteq D_n} \Pr(D_n^i) \cdot C(D_n^i)$$

où $C(D_n^i)$ représente le coût d'une donnée quelconque de D_n^i. Les séries génératrices (voir [Froidevaux-AL1]) constituent un outil mathématique puissant pour calculer la complexité des algorithmes.

Attention
Ce n'est parce qu'un algorithme est meilleur en moyenne qu'un autre en moyenne, qu'il est meilleur dans le pire des cas.

2.4 Propriété des complexités

Propriété
La complexité en moyenne et les complexités extrémales sont liées par la relation :

$$T_{min}(n) \leq T_{moy}(n) \leq T_{max}(n)$$

Remarque
Si le comportement de l'algorithme ne dépend pas de la configuration des données, ces trois quantités sont confondues. Mais en général, ce n'est pas le cas et l'on ne sait pas si le coût moyen est plus proche du coût minimal ou du coût maximal (sauf si l'on sait déterminer les fréquences relatives des configurations donnant un coût minimal et celles donnant un coût maximal).

En pratique
On ne s'intéresse qu'à la complexité au pire et à la complexité en moyenne.

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

2.5 Notion d'optimalité

Algorithme optimal
Un algorithme est dit **optimal** si sa complexité est la complexité minimale parmi les algorithmes de sa classe.

Exemple
On peut montrer que tout algorithme résolvant le problème du tri a une complexité dans le pire des cas en $\Omega(n \lg n)$. Le tri par tas (*heapsort*) est en $O(n \lg n)$ dans le pire des cas : il est donc optimal.

3 Exemple : Complexités de la recherche linéaire

Cette section illustre le calcul des complexités de la recherche linéaire d'un élément `x` dans un tableau `t` de `n` éléments.

Algorithme (à analyser)

```
Fonction rechseq ( t : Element [ NMAX ] ; n : Entier ; x : Element ) : Entier
Début
  |  j <- 1
  |  TantQue ( j <= n Et t [ j ] <> x ) Faire
  |    |  j <- j + 1
  |  FinTantQue
  |  Si j > n Alors
  |    |  Retourner ( - 1 )
  |  Sinon
  |    |  Retourner ( j )
  |  FinSi
Fin
```

Choix des opérations significatives

Dans cet algorithme, le paramètre reflétant la taille des données est n et les opérations significatives sont les comparaisons de `x` avec les éléments du tableau. En particulier, on ne tient pas compte des comparaisons de `j` avec `n`.

Complexités extrêmes

Le nombre de comparaisons varie avec la configuration du tableau, c.-à-d. la façon dont son contenu est organisé. En effet :

- Si `x` correspond à l'élément en `t[1]` : 1 comparaison.
- Si `x` correspond à l'élément en `t[n]` ou s'il n'apparaît pas dans le tableau : n comparaisons.
- Dans les autres cas : entre 1 et n comparaisons selon l'emplacement dans le tableau de l'élément correspondant à `x`.

Remarque

Dans le cas d'une recherche avec échec (l'élément cherché n'est pas dans le tableau), la complexité de l'algorithme est donc de n. Dans le cas d'une recherche avec succès (`x` est dans le tableau) la complexité de l'algorithme varie selon la configuration des données dans le tableau. On a donc :

$$T_{min}(n) = 1 \qquad \text{et} \qquad T_{max}(n) = n$$

Complexité en moyenne

Pour calculer $T_{moy}(n)$, on note q la probabilité d'avoir `x` dans le tableau, et on suppose que si `x` est dans le tableau, alors toutes les places sont équiprobables.

On note D_n^i pour $1 \leq i \leq n$, l'ensemble des données où `x` est à la i-ème place et D_n^0 l'ensemble des données où `x` est absent. D'après les conventions, on a :

$$\Pr(D_n^i) = \frac{q}{n} \qquad \text{et} \qquad \Pr(D_n^0) = 1 - q$$

D'après l'analyse de l'algorithme, on a aussi :

$$C(D_n^i) = i \qquad \text{et} \qquad C(D_n^0) = n$$

Par conséquent :

$$\begin{aligned} T_{moy}(n) &= \sum_{i=0}^{n} \Pr(D_n^i) \cdot C(D_n^i) \\ &= (1-q) \cdot n + \sum_{i=1}^{n} \frac{q}{n} \cdot i \\ &= (1-q) \cdot n + \frac{q}{2n} \cdot n \cdot (n+1) \\ &= (1-q) \cdot n + \frac{q}{2} \cdot (n+1) \end{aligned}$$

Si on sait que `x` est dans le tableau, on a $q = 1$ d'où :

$$T_{moy}(n) = \frac{n+1}{2}$$

Si `x` a une chance sur deux d'être dans le tableau, on a $q = \frac{1}{2}$ d'où :

$$T_{moy}(n) = \frac{n}{2} + \frac{1}{4}(n+1) = \frac{1}{4}(3n+1)$$

4 Étude : Sous-Séquence de somme maximale

Cette étude de cas étudie le **problème de la sous-séquence de somme maximale**. Ce problème est intéressant parce qu'il existe nombre d'algorithmes pour le résoudre et la complexité (en nombre d'opérations de somme) de ces algorithmes varie considérablement.

Problème de la sous-séquence de somme maximale

Étant donné un tableau `t[1..n]` d'entiers (positifs et négatifs), déterminer la **valeur maximale** du sous-tableau `t[g..h]` donnant la plus grande somme de tous les sous-tableaux contigus de `t`. Pour plus de commodité, la sous-séquence de somme maximale est 0 si tous les entiers sont négatifs.

Exemple

Si la séquence est [11,13,-4,3,-26,7,-13,25,-2,17,5,-8,1] alors la sous-séquence [3,-26,7,-13,25] a pour somme -4 et la sous-séquence de somme maximale est [25,-2,17,5] de somme 45.

4.1 Algorithme naïf

L'algorithme naïf adopte la plus simple des stratégies : évaluer la somme de chaque sous-tableau (parmi les $n(n+1)/2$ sous-tableaux possibles) et à chaque évaluation mémoriser la **meilleure somme**. Dans l'algorithme :

- La procédure `xactualiserSi` actualise `vmax` avec `somme` si `somme > vmax`
- La fonction `vmaxsomme1` est l'algorithme naïf du calcul de la « meilleure somme »

Algorithme naïf

```
Action xactualiserSi ( DR vmax : Entier ; somme : Entier )
Début
  |  Si ( somme > vmax )
  |    |  vmax <- somme
  |  FinSi
Fin
```

```
Fonction vmaxsomme1 ( DR t : Entier [ TMAX ] ; n : Entier ) : Entier
Variable vmax : Entier
Variable somme : Entier
Variable ix , jx , k : Entier
Début
  |  vmax <- t [ 1 ]
  |  Pour ix <- 1 à n Faire
  |    |  Pour jx <- ix à n Faire
  |    |    |  somme <- 0
  |    |    |  Pour k <- ix à jx Faire
  |    |    |    |  somme <- somme + t [ k ]
  |    |    |  FinPour
  |    |    |  xactualiserSi ( vmax , somme )
```

```
  |    |  FinPour
  |  FinPour
  |  Retourner ( vmax )
Fin
```

Complexité
Elle vaut :

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 = n(n+1)(n+2)/6 = \Theta(n^3)$$

Conclusion
Pour de grandes valeurs de n, la procédure `vmaxsomme1` est totalement inefficace.

4.2 Algorithme un peu moins naïf

La complexité de l'algorithme naïf provient de ses boucles imbriquées. Serait-il possible d'en enlever une ? En regardant de plus près, on s'aperçoit vite que l'on refait beaucoup de calculs plusieurs fois : le calcul de la somme d'une sous-séquence utilise les calculs d'une sous-séquence précédente. En d'autres termes, la boucle la plus interne est inutile car

$$\sum_{k=i}^{j} t[k] = \sum_{k=i}^{j-1} t[k] + t[j]$$

Il suffit donc d'ajouter `t[j]` à l'ancienne somme. Dans l'algorithme :

- La procédure `xcalculerDroite` calcule vers la droite la somme max de `t[ideb..ifin]`.
- La fonction `vmaxsomme2` est l'algorithme amélioré du calcul de la « meilleur somme ».

Algorithme amélioré

```
Action xcalculerDroite ( DR t : Entier [ TMAX ] ; ideb , ifin : Entier ; DR vmax :
    Entier )
Variable somme : Entier
Variable jx : Entier
Début
  |  somme <- 0
  |  Pour jx <- ideb à ifin Faire
  |    |  somme <- somme + t [ jx ]
  |    |  xactualiserSi ( vmax , somme )
  |  FinPour
Fin
```

```
Fonction vmaxsomme2 ( DR t : Entier [ TMAX ] ; n : Entier ) : Entier
Variable vmax : Entier
Variable ix : Entier
Début
  |  vmax <- t [ 1 ]
  |  Pour ix <- 1 à n Faire
  |    |  xcalculerDroite ( t , ix , n , vmax )
```

```
  |  FinPour
  |  Retourner ( vmax )
Fin
```

Complexité

Elle vaut :

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = n(n+1)/2 = \Theta(n^2)$$

Conclusion

Les médiocres performances des procédures `vmaxsomme1` et `vmaxsomme2` découlent de la manière de consulter les sous-tableaux : dans les deux cas, les $n(n+1)/2$ sous-tableaux sont explicitement formés par la double boucle :

```
| Pour ix <- 1 à n faire
| | Pour jx <- ix à n faire
| | | ...
```

d'où une complexité d'au moins $\Theta(n^2)$. Dès lors, pour réduire significativement le temps de recherche, il faut adopter une stratégie qui permet le traitement de tous les sous-tableaux tout en évitant de les construire de manière exhaustive.

4.3 Algorithme linéaire

Peut-on aller plus loin ? Peut-on encore enlever une boucle ? La réponse est oui mais la solution est peut-être moins triviale.

L'idée reste cependant la même : supprimer des calculs inutiles. Mais elle utilise ici le fait que l'on cherche un maximum.

Si donc on trouve une sous-séquence initiale de somme inférieure (ou égale) à 0, on peut supprimer cette sous-séquence initiale car elle apporte moins à la somme totale que de commencer juste après elle. Exemple : Dans la séquence [4,-5,3,...] il vaut mieux commencer au 3 (avec une somme initiale qui vaut 0) que commencer au 4 (qui nous donne une somme de -1 arrivé au 3). L'algorithme est le suivant :

Algorithme optimal

```
Fonction vmaxsomme4 ( DR t : Entier [ TMAX ] ; n : Entier ) : Entier
Variable vmax : Entier
Variable somme : Entier
Variable ix , jx : Entier
Début
  |  ix <- 1
  |  vmax <- t [ ix ]
  |  somme <- 0
  |  Pour jx <- 1 à n Faire
  |    |  Si ( somme < 0 ) Alors
  |    |    |  somme <- 0
```

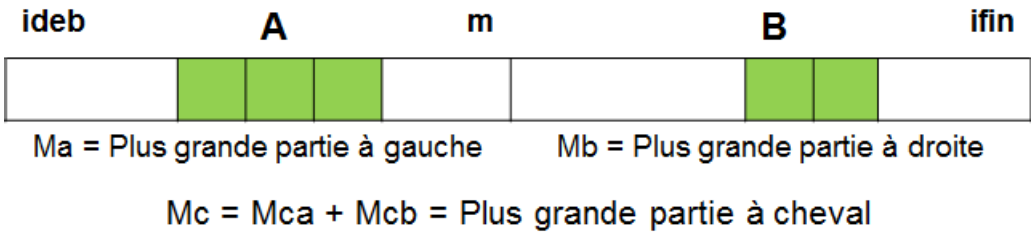
```
  |    |    |  ix <- jx
  |    |  FinSi
  |    |  somme <- somme + t [ jx ]
  |    |  xactualiserSi ( vmax , somme )
  |  FinPour
  |  Retourner ( vmax )
Fin
```

Complexité
Elle vaut :

$$\sum_{i=1}^{n} 1 = n = \Theta(n)$$

4.4 Algorithme récursif

Le tableau initial est scindé en deux parties de tailles à peu près égales (selon que n est pair ou impair) : la plus grande somme se trouve soit dans le sous-tableau `B` de droite, soit dans le sous-tableau `A` de gauche, soit à cheval sur les deux sous-parties. Dans ce cas elle est constituée d'une plus grande somme de la partie gauche se terminant à la fin de la partie gauche (c.-à-d. en `m`), et d'une plus grande somme de la partie droite commençant au début de la partie droite (c.-à-d. en `m+1`).



La procédure est récursive. Pour « sortir » des appels récursifs, il est nécessaire de rencontrer un « couple de données-paramètres » (transmis à l'appel) dont la solution est triviale. C'est le cas si le tableau est composé d'au plus un élément. L'algorithme est le suivant :

- La procédure `xcalculerGauche` calcule vers la gauche la somme max de `t[ideb..ifin]`.
- La procédure `vmaxsomme3Rec` est l'algorithme récursif du calcul de la « meilleur somme ».
- La fonction `vmaxsomme3` est la fonction maître.

Algorithme récursif

```
Action xcalculerGauche ( DR t : Entier [ TMAX ] ; ideb , ifin : Entier ; DR vmax :
    Entier )
Variable somme : Entier
Variable jx : Entier
Début
```

```
  |  somme <- 0
  |  Pour jx <- ifin à ideb Pas - 1 Faire
  |    |  somme <- somme + t [ jx ]
  |    |  xactualiserSi ( vmax , somme )
  |  FinPour
Fin
```

```
Fonction vmaxsomme3 ( DR t : Entier [ TMAX ] ; n : Entier ) : Entier
Variable vmax : Entier
Début
  |  vmaxsomme3Rec ( t , 1 , n , vmax )
  |  Retourner ( vmax )
Fin
```

```
Action vmaxsomme3Rec ( DR t : Entier [ TMAX ] ; ideb , ifin : Entier ; R vmax : Entier )
Variable milieu : Entier
Variable gmax : Entier
Variable dmax : Entier
Variable mgmax , mdmax : Entier
Début
  |  Si ( ideb = ifin ) Alors
  |    |  vmax <- t [ ideb ]
  |  Sinon
  |    |  milieu <- DivEnt ( ideb + ifin , 2 )
  |    |  mgmax <- t [ milieu ]
  |    |  xcalculerGauche ( t , ideb , milieu , mgmax )
  |    |  mdmax <- t [ milieu + 1 ]
  |    |  xcalculerDroite ( t , milieu + 1 , ifin , mdmax )
  |    |  vmax <- mgmax + mdmax
  |    |  vmaxsomme3Rec ( t , ideb , milieu , gmax )
  |    |  vmaxsomme3Rec ( t , milieu + 1 , ifin , dmax )
  |    |  xactualiserSi ( vmax , gmax )
  |    |  xactualiserSi ( vmax , dmax )
  |  FinSi
Fin
```

Complexité

Elle est donnée par la récurrence :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{sinon} \end{cases}$$

Utilisons le Master-théorème du module @[Algorithmes diviser pour régner] : ici $a = 2$ et $b = 2$ donc $\log_b a = 1$ et nous nous trouvons dans le cas 2 du théorème :

$$f(n) = \Theta(n^{\log_b a}) = \Theta(n)$$

Par conséquent :

$$T(n) = \Theta(n \lg n)$$

4.5 Tests des algorithmes

Algorithme principal

```
Constante TMAX <- 50

Algorithme pgvmaxsomme1
Variable tab : Entier [ TMAX ]
Variable nelems : Entier
Variable vmax1 : Entier
Variable vmax2 : Entier
Variable vmax3 : Entier
Variable vmax4 : Entier
Début
  |  saisirTab ( tab , nelems )
  |  vmax1 <- vmaxsomme1 ( tab , nelems )
  |  vmax2 <- vmaxsomme2 ( tab , nelems )
  |  vmax3 <- vmaxsomme3 ( tab , nelems )
  |  vmax4 <- vmaxsomme4 ( tab , nelems )
  |  Afficher ( "maxsomme1 de tab: " , vmax1 : 5 )
  |  Afficher ( "maxsomme2 de tab: " , vmax2 : 5 )
  |  Afficher ( "maxsomme3 de tab: " , vmax3 : 5 )
  |  Afficher ( "maxsomme4 de tab: " , vmax4 : 5 )
Fin
```

5 Conclusion

Le choix et l'application de méthodes rigoureuses sont indispensables pour l'élaboration d'algorithmes efficaces. La performance des programmes doit être le plus possible indépendant du matériel. Elle doit dépendre avant tout de la manière de concevoir et de spécifier les algorithmes.

6 Références générales

L'analyse du temps d'exécution d'algorithmes a été rendu populaire par Knuth dans [Knuth-AL1], [Knuth-AL2] et [Knuth-AL3]. Les notations Grand-Oh, Grand-Oméga, Grand-Theta ont été préconisées par Knuth dans [Knuth-R1].

Le problème de la sous-séquence de somme maximum est de [Bentley-AL2].

La série de livres [Bentley-AL1], [Bentley-AL2] et [Bentley-AL3] montrent comment optimiser les programmes pour la vitesse.

Aho-AL1 @BookAho-AL1, author = Aho, A.V. AND Hopcroft, J.E. AND Ullmann, J.D., title = The Design and Analysis of Computer Algorithms, publisher = Addison-Wesley, series = Reading, Mass., year = 1974

Bentley-AL1 @BookBentley-AL1, author = Bentley J. L., title = Writing Efficient Programs, publisher = Prentice Hall, Englewood Cliffs, N.J., ISBN = , year = 1982

Bentley-AL2 @BookBentley-AL2, author = Bentley J. L., title = Programming Pearls, publisher = Addison-Wesley, Reading, Mass., ISBN = , year = 1986

Bentley-AL3 @BookBentley-AL3, author = Bentley J. L., title = More Programming Pearls, publisher = Addison-Wesley, Reading, Mass., ISBN = , year = 1988

Froidevaux-AL1 @BookFroidevaux-AL1, author = Froidevaux, Ch. AND Gaudel, M.-C. AND Soria, M., title = Types de données et algorithmes, publisher = Ediscience, year = 1994, note = Algorithmique fondamentale – Pseudo-code

Knuth-AL1 @BookKnuth-AL1, author = Knuth D.E., title = Fundamental Algorithms, publisher = Addison-Wesley, Reading, Mass., year = 1973, type = , series = The Art of Computer Programming, volume = 1 edition = 2, note = Ouvrage de référence

Knuth-AL2 @BookKnuth-AL2, author = Knuth Donald E., title = Seminumerical Algorithms, publisher = Addison-Wesley, Reading, Mass., year = 1981, isbn = , type = , series = The Art of Computer Programming, volume = 2 edition = , school = , note = Ouvrage de référence

Knuth-AL3 @BookKnuth-AL3, author = Knuth Donald E., title = Sorting and searching, publisher = Addison-Wesley, Reading, Mass., year = 1975, isbn = , type = , series = The Art of Computer Programming, volume = 3 edition = , school = , note = Ouvrage de référence

Knuth-R1, 18-23 @InProceedingsKnuth-R1, author = Knuth Donald E., title = Big Omicron and Big Omega and Big Theta, booktitle = ACM SIGACT News 8 (1976),