

Complexité et Algorithmique avancée

PLAN

1. Introduction
2. Les bases de l'analyse d'un algorithme
 - Problème, instance d'un problème, programme
 - L'analyse d'un algorithme
 - Calcul de la complexité d'un algorithme
 - Les types de complexité
 - La complexité asymptotique
 - Les notations de Landau
 - Validation d'un algorithme
3. Outils mathématiques
 - Sommations
 - Récurrences
 - Méthodes de résolution des équations de récurrence
4. Structures de données et comparaison d'algorithmes
 - Comparaison des algorithmes de tri
 - Les arbres
 - Les tables de hachage
5. Les classes de problèmes

1. Introduction

En informatique, deux questions fondamentales se posent toujours devant un problème à résoudre :

- a. existe-t-il un algorithme pour résoudre le problème en question ?
- b. si oui, cet algorithme est-il utilisable en pratique, autrement dit le temps et l'espace mémoire qu'il exige pour son exécution sont-ils "raisonnables" ?

La première question a trait à la calculabilité, la seconde à la complexité.

1.1. La recherche en logique et en mathématique : bref historique

En calculabilité on cherche à définir les problèmes dont les solutions sont calculables par un algorithme et ceux qui ne le sont pas. Ces derniers sont les problèmes pour lesquels il n'existe pas de solution (algorithme) quel que soit les progrès futurs de la technologie.

La théorie de la calculabilité est née à la fin du 19^{ème} siècle de questions de logique et de mathématique. On cherchait à connaître les limites de la connaissance et la prouvabilité d'une vérité mathématique donnée.

Résoudre un problème en logique revient à donner une démonstration d'une proposition quelconque. Dans le cadre du calcul propositionnel, on disposait depuis 1934 de méthodes de preuves de validité (les systèmes de déduction naturelle de Gentzen) ce qui établit la décidabilité du calcul propositionnel.

Au début du 20^{ème} siècle, **David Hilbert** avait pour projet de rechercher les principes de base des mathématiques. Il se propose de :

- a. formaliser toutes les mathématiques
- b. trouver les axiomes et les règles de déductions qui permettraient de démontrer toutes les vérités mathématiques
- c. mécaniser (automatiser) le raisonnement mathématique.

Mais en 1931, le mathématicien **Kurt Gödel** publie son théorème d'incomplétude qui démontre les limites de ce projet : "dans n'importe quel système formel, il existe une proposition indécidable", autrement dit, une proposition pour laquelle on ne saura pas démontrer ni sa vérité ni sa fausseté.

Le projet de Hilbert s'avère donc être un projet impossible. Ainsi :

- a. tout n'est pas démontrable ; et
- b. on ne peut pas donner une liste finie de tous les principes permettant de développer une preuve mathématique.

1.2. Conséquences du théorème d'incomplétude sur l'informatique

D'un point de vue informatique, le théorème d'incomplétude de Gödel signifie une impossibilité fondamentale. Tout problème ne peut être résolu par un algorithme. Il est donc impossible de concevoir un programme capable de vérifier tous les programmes. Pour prouver ce résultat, on suppose qu'un programme P capable de vérifier tous les programmes existe et on définit un programme B et on montre que P ne peut pas le vérifier.

Preuve :

Supposons qu'un programme capable de vérifier tous les programmes existe. Soit P ce programme.

- a. Pour tout programme A, s'il est juste alors on écrit que $P(A) = v$ (« vrai ») et s'il est faux on écrit $P(A) = f$ (« faux »).
- b. Soit B le programme suivant : $B = \text{« } P(B) = f \text{ »}$.
- c. - Si $P(B) = v$ (le programme B est juste), alors le programme $P(B) = f$ est faux ; donc $P[P(B) = f] = P(B) = f$, ce qui est contraire à l'hypothèse.
- Si $P(B) = f$, alors on a $P[P(B) = f] = P(B) = v$, ce qui est encore contraire à l'hypothèse.
- d. Ainsi B ne peut pas être vérifié par P. Il ne peut donc pas exister de programme capable de vérifier tous les programmes.

Avec l'apparition des ordinateurs, en décidabilité, on s'intéresse à étudier les limites de l'informatique indépendamment des machines qui les exécutent. Il s'agit de trouver quels sont les problèmes qui sont insolubles (ou indécidables) et qui le seront toujours quelque soit le développement technologique futur.

Pour les problèmes décidables, une seconde nécessité apparaît : il n'était plus suffisant de savoir qu'une solution existe (la décidabilité du problème), mais il devenait impératif de savoir que la solution soit réalisable, "exécutable" de manière "raisonnable". C'est le domaine de la complexité des algorithmes ou encore l'analyse des algorithmes.

Ce théorème implique ainsi une dichotomie entre les problèmes qui ont une solution algorithmique et ceux qui n'en ont pas.

La recherche s'est développée par la suite dans ces deux directions. Pour les problèmes qui ont une solution algorithmique, on cherchera à les comparer et les classer en une hiérarchie en fonction de leur complexité, une fonction qui mesure les ressources (temps ou espace) nécessaires à l'exécution de l'algorithme en question.

Certes on peut définir des méthodes de vérification efficaces et les outiller de sorte qu'elles soient faciles à utiliser. Mais ces méthodes, aussi efficaces soient-elles, ne garantissent pas que tous les programmes qu'elles valident soient corrects.

La ressource temps mesure le nombre d'opérations élémentaires nécessaires en fonction de la taille des données. La taille des données s'exprime en nombre de caractères nécessaire pour décrire une donnée du problème.

Généralement le temps de calcul d'un algorithme croît avec la taille des données. La complexité est la vitesse de cette croissance.

Une croissance exponentielle rend un problème intraitable pour les données de grande taille. Une croissance polynomiale, avec une puissance au dessus des puissances 3 ou 4 rend aussi le problème intraitable pour des données de grande taille.

Exemple :

Générer toutes les permutations de n éléments. La complexité est plus qu'exponentielle puisqu'il y a $n!$ permutations à faire. Si n est grand le temps est énorme !

La complexité est en quelque sorte la maîtrise de l'algorithmique. Cette science vise à déterminer, sous un certain nombre de conditions, quel sont les temps de calcul et espace mémoire requis pour obtenir un résultat donné. Une bonne maîtrise de la complexité se traduit par des applications qui tournent en un temps prévisible et sur un espace mémoire contrôlé. A l'inverse, une mauvaise compréhension de la complexité débouchent sur des latences importantes dans les temps de calculs, ou encore des débordements mémoire conséquents, qui au mieux, "gèlent" les ordinateurs ("swap" sur le disque dur) et au pire font carrément "planter" la machine.

2. Les bases de l'analyse d'un algorithme

Le but en analyse d'algorithmes est d'étudier le fonctionnement d'un algorithme. Il s'agit, pour les problèmes solubles, d'estimer les ressources nécessaires au déroulement de la solution considérée. Les principales ressources qui nous intéressent sont les fonctions temps d'exécution et espace mémoire nécessaire. Pour formaliser ces notions on doit distinguer la notion de problème de celle de programme.

2.1. Problème, instance, solution et programme

Le but en analyse d'algorithmes est d'étudier le fonctionnement d'un algorithme. Il s'agit, pour les problèmes solubles, d'estimer les ressources nécessaires au déroulement de la solution considérée. Les principales ressources qui nous intéressent sont les fonctions temps d'exécution et espace mémoire nécessaire. Pour formaliser ces notions on doit distinguer la notion de problème de celle de programme.

2.1.1. Notion de problème

Quelles sont les caractéristiques d'un problème?

- Un problème est une question "générique", c'est-à-dire qu'un problème contient des paramètres ou variables libres. Lorsque l'on attribue une valeur à ces variables libres on obtient une instance du problème;
- Un problème existe indépendamment de toute solution ou de la notion de programme pour le résoudre;
- Un problème peut avoir plusieurs solutions, plusieurs algorithmes différents peuvent résoudre le même problème.

Exemples :

- a. Déterminer si un entier donné est pair ou impair ;*
- b. déterminer le maximum d'un ensemble d'entiers donnés*
- c. trier une suite d'entiers donnés, etc.*

2.1.2. Notion d'instance

Une instance de problème est la question générique appliquée à un élément :
Maintenant que nous avons une idée intuitive et précise de ce qu'est un problème essayons de préciser quelque peu la notion d'algorithme (ou encore de programme).

Exemples :

- a. Déterminer si un entier donné est pair ou impair ;*
- b. Déterminer le minimum d'un ensemble d'entiers donnés ;*
- c. Trier une suite d'entiers donnés, etc.*

2.1.3. Solution

Définition : Une procédure effective est définie en termes de langage décidé (accepté) par une machine de Turing. Pour démontrer que certains problèmes ne sont pas solubles par une procédure effective, on prouve que les langages qu'ils encodent ne sont pas décidés par une machine de Turing.

Une solution à un problème pouvant être exécuté par un ordinateur est une procédure effective.

Si un problème est récursif alors il existe une procédure qui le résout. Mais rien ne garantit que cette procédure soit utilisable pratiquement : le temps et l'espace mémoire nécessaire à son exécution pourraient largement dépasser ce dont on peut espérer disposer. Un algorithme utilisable se doit d'être efficace.

2.1.4. Notion de programme

Un programme c'est la solution d'un problème pouvant être exécutée par un ordinateur. Il contient toute l'information nécessaire pour résoudre le problème en temps fini.

Cette notion va permettre de distinguer les solutions qui pourront être exécutées par un ordinateur ou "procédure effective" des solutions non effectives (qui ne s'arrêtent pas).

Un programme C est une procédure effective, pourquoi ? Un code C peut être compilé et ensuite exécuté "mécaniquement" par le processeur. Une autre caractéristique essentielle d'une procédure effective est qu'elle contient exactement la marche à suivre pour résoudre le problème et qu'aucune décision supplémentaire ne doit être prise lors de l'exécution de la procédure.

Pour qu'une procédure soit considérée comme effective pour résoudre un problème, il faut que celle-ci se termine sur toutes les instances du problème.

L'exemple suivant donne une idée d'une procédure non effective.

Exemple : Déterminer si un programme donné n'a pas de boucles ou de séquences d'appels récursifs infinies.

Il n'y a aucun moyen pour savoir si une boucle est infinie ou non.

2.2. L'analyse d'un algorithme

L'analyse des algorithmes s'exprime en termes d'évaluation de la complexité de ces algorithmes. On présente dans cette section les éléments de base de la complexité.

2.2.1. Définition d'un algorithme

- Procédure de calcul bien définie qui prend en entrée un ensemble de valeurs et produit en sortie un ensemble de valeurs. (Cormen, Leiserson, Rivert)
- Ensemble d'opérations de calcul élémentaires, organisé selon des règles précises dans le but de résoudre un problème donné. Pour chaque donnée du problème, il retourne une réponse après un nombre fini d'opérations.
(Beauquier, Berstel, Chrétienne)
- Spécification d'un schéma de calcul sous forme d'une suite finie d'opérations élémentaires obéissant à un enchaînement déterminé.
(Al Khowarizmi, Bagdad IX^e siècle. *Encyclopedia Universalis*):

Un algorithme doit satisfaire les propriétés suivantes :

- a. Il peut avoir zéro ou plusieurs quantités de données en entrée ;
- b. Il doit produire au moins une quantité en sortie ;
- c. Chacune de ses instructions doit être claire et non ambiguë ;
- d. Il doit terminer après un nombre fini d'étapes ;
- e. Chaque instruction doit être implémentable ;

Exemple : Soit le problème : "trier un ensemble de $n \geq 1$ d'entiers".

Une solution pourrait être la suivante :

"Parmi les entiers qui ne sont pas encore triés, rechercher le plus petit et le placer comme successeur dans la liste triée. La liste triée est initialement vide".

Cette solution n'est pas un algorithme car elle laisse plusieurs questions sans réponse. Elle ne dit pas où ni comment les entiers sont initialement triés ni où doit être placé le résultat.

On supposera que les entiers sont dans un tableau t tel que le $i^{\text{ème}}$ entier soit stocké à la $i^{\text{ème}}$ position : $t[i]$; $0 \leq i < n$. L'algorithme suivant est la solution d'un tri par sélection.

```
for (i=0; i<n; i++) {  
    calculer le min de  $t$ , de  $i$  à  $n-1$ , et supposer que le plus petit est en  $t[\text{min}]$ ;  
    permuter  $t[i]$  et  $t[\text{min}]$ ;  
}
```

En algorithmique (étude des algorithmes), en dehors de l'analyse, on peut s'intéresser aux questions suivantes :

- la modularité ;
- la correction ;
- la maintenance ;
- la simplicité ;
- la robustesse ;
- l'extensibilité,
- le temps de mise en œuvre,
- etc.

2.2.2. Qu'est ce que l'analyse ?

L'analyse d'un algorithme consiste à déterminer la quantité de ressources nécessaires à son exécution. Ces ressources peuvent être la quantité de mémoire utilisée, la largeur d'une bande passante, le temps de calcul etc.

Nous nous intéressons dans ce cours au temps et à la mémoire. Le but est de pouvoir identifier, face à plusieurs algorithmes qui résolvent un même problème, celui qui est le plus efficace.

L'étude des performances des algorithmes a plusieurs avantages :

- l'algorithmique nous aide à comprendre la scalabilité (portabilité sur des plates-formes de taille plus réduite ou plus grande)
- les performances limitent une frontière entre ce qui est faisable et ce qui est impossible à faire ;
- Les mathématiques de l'algorithmique fournissent un langage pour parler du comportement des programmes ;
- les conséquences tirées des performances des programmes peuvent se généraliser à d'autres ressources informatiques ;
- les performances sont le crédit de l'informatique ;
- la rapidité est une performance recherchée dans la plupart de nos activités, elle a un caractère agréable !

2.3. Les paramètres de l'analyse

Les performances d'un algorithme dépendent de trois principaux paramètres suivants:

- ***La taille des entrées***

Il est intuitivement évident, par exemple, que les séquences courtes sont plus faciles à trier que les séquences longues. La complexité est donc paramétrée par la taille des données puisque, ce paramètre taille dépend du problème étudié, mais très souvent c'est le nombre d'éléments constituant l'entrée. C'est par exemple la longueur d'une liste pour un algorithme de recherche ou de tri, le nombre total de bits nécessaire pour représenter l'entrée pour un produit de matrices. Ce paramètre peut être composé de plus d'une valeur. Pour un graphe, l'entrée est le nombre de nœuds et le nombre d'arcs. Pour chaque problème il faut en premier lieu caractériser l'entrée.

- ***La distribution des données***

Un tableau déjà trié est facile à trier et un tableau trié en sens inverse de l'ordre cherché est le plus long à réaliser. Mais cette information est généralement difficile à obtenir.

- ***La structure de données***

La représentation des données un paramètre fondamental. Une donnée directement accessible (l'élément d'un tableau) est plus rapidement obtenue qu'un élément dans une structure à accès séquentiel comme les listes.

2.4. Les types de complexité

Il existe trois types de complexité :

- ***complexité du meilleur cas***

C'est le cas, pour l'exemple du tri, où la suite initiale est triée. Cette mesure n'a pas d'intérêt puisque tous les algorithmes réagissent de la même manière. Elle ne permet pas de distinguer deux solutions.

- ***complexité en moyenne***

Elle nécessite la connaissance de la distribution des données autrement dit des fonctions de probabilités sur les données, qui peuvent être obtenues après avoir effectué plusieurs tests.

- ***complexité du cas pire***

Elle fournit une borne supérieure du temps d'exécution qui indique que dans toutes les situations, l'algorithme ne peut pas dépasser la valeur de cette borne. Elle constitue donc une garantie, ce que nous cherchons toujours ! C'est donc à cette complexité que nous nous intéressons dans la suite de ce cours.

Il s'agit donc de trouver une équation qui relie le temps d'exécution à la taille des données. Nous pourrions ainsi comparer deux algorithmes qui résolvent le même problème en comparant le rapport de leur équation et choisir le plus rapide des deux.

2.5. La complexité asymptotique

En pratique, il est difficile de calculer de manière exacte la complexité d'un algorithme. Si n est la donnée d'entrée, on se limite à une évaluation asymptotique de l'algorithme. La complexité asymptotique est une approximation du nombre d'opérations que l'algorithme exécute en fonction de la donnée entrée: asymptotique car elle prend en compte une donnée de grande taille et ne retient que le terme de poids fort dans la formule et ignore le coefficient multiplicateur.

Exemple Soit n la donnée de grande taille et $T(n)$ la complexité telle que $T(n) = 10*n^3 + 3*n^2 + 5*n + 1$, la complexité asymptotique est : $T(n) = O(n^3)$.

2.5.1. Les notations Landau

Les notations de Landau portent le nom du mathématicien allemand spécialisé en théorie des nombres Edmund Landau qui utilisa les notations introduites primitivement par Paul Bachmann.

- La notation O (grand O)
- La notation o (petit o)
- La notation Ω (grand oméga)
- La notation ω (petit oméga)
- La notation Θ (grand theta)
- La notation \sim (de l'ordre de ; équivalent à)

(Remarque : On va s'intéresser principalement aux notations O , Ω et Θ)

a. Notation grand O :

Soit $f(n)$ le nombre d'opérations effectués par un algorithme ; on dit que le temps mis par l'algorithme est $O(g(n))$ ou bien $f(n) \in O(g(n))$ et s'il existe une constante positive c et il existe une constante n_0 telle que $f(n) \leq cg(n) \quad \forall n \geq n_0$.

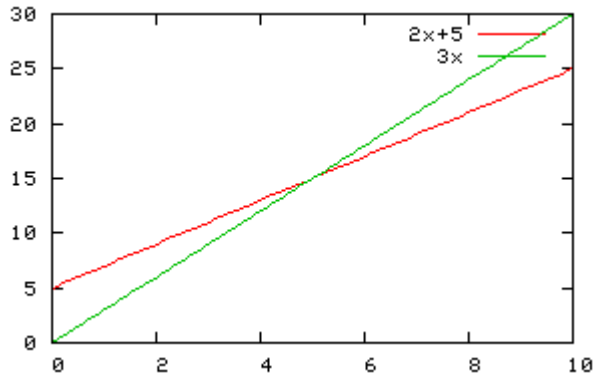
On dit que $g(n)$ est une borne supérieure asymptotique pour $f(n)$, on note abusivement $f(n) = O(g(n))$.

Formellement :

$$O(g(n)) = \{ f : \mathbb{N} \rightarrow \mathbb{N} \mid \exists c > 0 \text{ et } n_0 \geq 0 \text{ tels que } 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0 \}$$

Exemple : $f(n) = 2n+5 = O(n)$ car $2n+5 \leq 3n \quad \forall n \geq 5$ d'où $c=3$ et $n_0=5$

Le graphe ci-dessous illustre l'exemple :



b. Notation Ω

$$\Omega(g(n)) = \{ f : \mathbb{N} \rightarrow \mathbb{N} \mid \exists c > 0 \text{ et } n_0 \geq 0 \text{ tels que } f(n) \geq cg(n) \geq 0 \quad \forall n \geq n_0 \}$$

Si $f(n) \in \Omega(g(n))$, on dit que $g(n)$ est une borne inférieure asymptotique pour $f(n)$, on note abusivement $f(n) = \Omega(g(n))$.

Exemple : $f(n) = 2n+5 = \Omega(n)$ car $2n+5 \geq 2n \quad \forall n \geq 1$ d'où $c=2$ et $n_0=0$

c. Notation Θ

$$\Theta(g(n)) = \{ f : \mathbb{N} \rightarrow \mathbb{N} \mid \exists c_1 > 0, \exists c_2 > 0 \text{ et } n_0 \geq 0 \text{ tels que } c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0 \}$$

Si $f(n) \in \Theta(g(n))$ On dit que $g(n)$ est une borne asymptotique pour $f(n)$, on note abusivement $f(n) = \Theta(g(n))$

Exemple : $f(n) = 2n+5 = \Theta(n)$ car $2n \leq 2n+5 \leq 3n \quad \forall n \geq 5$ d'où $c_1=2$, $c_2=3$ et $n_0=5$

Remarque : Notations o et ω avec des inégalités strictes.

2.5.2. Propriétés

- a. Réflexivité : $f(n) = O(f(n))$.
- b. Transitivité : si $f(n) = O(g(n))$ et $g(n) = O(h(n))$ alors $f(n) = O(h(n))$.
- c. Somme : si $f(n) = O(h(n))$ et $g(n) = O(h(n))$ alors $f(n) + g(n) = O(h(n))$.
- d. Produit : si $f(n) = O(F(n))$ et $g(n) = O(G(n))$ alors $f(n) \times g(n) = O(F(n) \times G(n))$; en particulier $cx f(n) = O(F(n))$ pour toute constante c , car toute fonction constante est $O(1)$.

Remarque : On a les mêmes propriétés avec Ω et Θ .

On peut aisément déduire les propriétés suivantes :

- Si $p(n)$ est un polynôme de degré k alors $p(n) = \Theta(n^k)$.
- $\log_a n = \Theta(\log_b n)$. Nous pouvons donc dire qu'un algorithme est de complexité $\log n$ sans avoir à spécifier la base.
- $\log(n+1) = \Theta(\log n)$ pour toute base.
- $(n+1)^k = O(n^k)$.

2.5.3. Vocabulaire

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité.

$O(1)$: temps constant, indépendant de la taille des données. C'est le cas le plus optimal. $O(1) = O(n^k)$ avec $k=0$.

Exemple : L'algorithme de résolution de l'équation du second degré.

$O(n^k)$: complexité polynômiale,

$O(n)$: linéaire (*exemple* : la recherche d'une valeur dans un tableau)

$O(n^2)$: quadratique (*exemple* : tri des éléments d'un tableau)

$O(n^3)$: cubique (*exemple* : produit de deux matrices)

...

$O(\log n)$, $O(n \log n)$, ... complexité logarithmique (*exemple* recherche binaire)

$O(2^n)$, $O(n!)$, ... : complexité exponentielle, factorielle, ...

Exemple : si $T(n) = 2^n$ pour $n=60 \Rightarrow T(n) = 1,15 \times 10^{12}$ secondes
 $1,15 \times 10^{12}$ secondes ≈ 36.558 ans et si $T(n) = n!$ on a $n! > 2^n$

Exercices : Montrer que

1. $f(n)=5n^2 + -n = \Theta(n^2)$
2. $f(n)=6n^3 \neq \Theta(n^2)$
3. $f(n)=n^2 = O(10^{-5} n^3)$

Résumé : On compare les algorithmes sur la base de leur complexité.

La fonction exponentielle est toujours plus forte que la fonction polynomiale qui est plus forte que la fonction linéaire qui est plus forte que la fonction logarithmique.

2.6 Règles de la notation O

Pour découvrir les règles de la notation O, nous allons voir quelques exemples importants.

Exemple 1 : Soit i de type entier

```
i := 1 ;  
i := 2 ;  
i := 3 ;  
i := 4 ;  
i := 5 ;
```

On notera $T(5)$ le temps d'exécution de l'algorithme, car celui-ci compte 5 opérations.

Exemple 2 : Initialisation d'un tableau A de n éléments :

```
Pour  $i := 1$  à  $n$  faire  
    Lire ( $A[i]$ ) ;  
fait ;
```

Cet algorithme va effectuer une opération par élément du tableau, soit n opérations au total. Le temps d'exécution associé à cet algorithme sera donc de $T(n)$. Le temps d'exécution est proportionnel au nombre d'éléments à traiter.

Ainsi, la complexité de cet algorithme est de **$O(n)$** .

Facteurs constants et addition :

La première règle à retenir est que ***tout facteur constant est simplifié à 1.***

Le but de la notation O n'est pas d'évaluer en détail les performances d'un algorithme mais de donner une ***idée globale*** du résultat, et pour ce faire, on ne considère que les facteurs évoluant rapidement. Les constantes n'évoluant pas en fonction du volume de données, elles peuvent être simplifiées.

Exemple 3 :

```
Ecrire('bonjour') ;  
Ecrire('donner le nombre d'éléments') ;  
Lire(n) ;  
Pour i :=1 à n faire  
    lire(A[i]) ;  
fait ;
```

Le temps d'exécution est $T(3+n)$. Néanmoins, l'addition en notation O ne conserve que l'élément le plus important. Ici nous avons d'une part $O(1)$ et de l'autre $O(n)$. Comme le tableau comportera plus de 3 éléments, la complexité de cet algorithme sera de : **$O(1) + O(n) = O(n)$.**

Règle de la multiplication :

La première règle de multiplication est que les constantes multiplicatives sont omises. Ainsi, si on initialise deux tableaux $A1$ et $A2$ (exemple 2 : deux boucles consécutives),

```
Pour i :=1 à n faire  
    Lire (A1[i]) ;  
fait ;  
Pour i :=1 à n faire  
    Lire (A2[i]) ;  
fait ;
```

la complexité sera $O(n+n)=O(2n)=O(n)$, et ce parce que le facteur n évolue plus vite que la constante 2.

2.7 Estimation du coût d'un algorithme :

Pour analyser un algorithme on se donne un modèle d'analyse. Généralement ce modèle est la machine à accès aléatoire (RAM). Dans ce modèle les instructions sont exécutées séquentiellement sans opérations simultanées. L'analyse fait appel à des outils mathématiques, comme l'algèbre, l'algèbre combinatoire discrète, la théorie élémentaire des probabilités etc.

Sachant qu'un algorithme peut se comporter différemment pour chaque valeur d'entrée possible, il est nécessaire de limiter ce comportement à quelques formules simples et faciles à comprendre. C'est le nombre d'opérations élémentaires sur une entrée particulière, (affectations, comparaisons, opérations arithmétiques) effectuées par l'algorithme.

La complexité d'un algorithme s'exprime en fonction de la taille n des données. On dit que la complexité de l'algorithme est $O(f(n))$ où f est d'habitude une combinaison de polynômes, logarithmes ou exponentielles. Ceci reprend la notation mathématique classique, et signifie que le nombre d'opérations effectuées est borné par $cf(n)$, où c est une constante, lorsque n tend vers l'infini.

Considérer le comportement à l'infini de la complexité est justifié par le fait que les données des algorithmes sont de grande taille et qu'on se préoccupe surtout de la croissance de cette complexité en fonction de la taille des données.

La recherche de l'algorithme ayant la plus faible complexité, pour résoudre un problème donné, fait partie du travail régulier de l'informaticien.

- 1) La complexité de l'instruction d'affectation, instruction de lecture, l'instruction d'écriture peut en général se mesurer par $O(1)$. Sauf l'appel d'une fonction par une affectation et aussi si l'affectation peut porter sur les tableaux de longueur quelconque.
- 2) La complexité d'une suite d'instructions est déterminée par la règle de la somme. Autrement dit, elle est égale, à une constante multiplicative près, la complexité la plus forte parmi toutes les instructions de la suite.

Exemple : considérons deux modules de complexité respectives $O_f(n)$ et $O_g(n)$ où :

$$f(n) = \begin{cases} n^4 & \text{si } n \text{ est pair} \\ n^2 & \text{si } n \text{ est impair} \end{cases} \quad g(n) = \begin{cases} n^2 & \text{si } n \text{ est pair} \\ n^3 & \text{si } n \text{ est impair} \end{cases}$$

La complexité est $O(\max(f(n), g(n)))$ c'est à dire n^4 si n est pair et n^3 si n est impair.

- 3) La complexité d'une instruction conditionnelle (si) est celle des instructions exécutées dans la condition, plus celle de l'évaluation de la condition. Cette dernière est en général égale à $O(1)$. La complexité de l'instruction (si-alors-sinon) correspond à la complexité d'évaluation de la condition plus la complexité la plus grande entre la série d'instructions exécutée si la condition est vraie, et celle exécutée si la condition est fausse.
- 4) La complexité d'une boucle est la somme cumulée sur toutes les itérations, de la complexité des instructions exécutées dans le corps de la boucle, plus l'évaluation de la condition (généralement $O(1)$). Assez souvent il s'agit du produit du nombre

d'itérations (s'il est évidemment fini) par la plus grande complexité rencontrée dans l'exécution de la boucle.

Exemple 1: *Calcul du maximum de 4 nombres entiers a, b, c, d.*

Solution 1 :

```
int maximum(int a, int b, int c, int d)
{
    int max=a ;
        if (b>max) max=b ;
        if (c>max) max=c;
        if (d>max) max=d;
    return max;
}
```

Solution 2 :

```
int maximum(int a, int b, int c, int d)
{
    if (a>b)
        if (a>c)
            if ( a>d) return a;
            else return d;
        else if (c>d) return c;
        else return d;
    if (b>c)
        if (b>d) return b;
        else return d;
    else if (c>d) return c;
    else return d;
}
```

Les deux solutions exigent exactement 3 comparaisons pour déterminer la réponse bien que la solution 1 soit plus simple à comprendre. Elles sont de même complexité temporelle pour une exécution sur machine. Mais en termes d'espace, la solution1 exige un espace supplémentaire pour stocker le max. La quantité d'espace supplémentaire étant insignifiante, ces deux solutions sont aussi équivalentes en complexité spatiale. L'évaluation de l'efficacité d'un algorithme en termes d'espace occupé et la comparaison entre algorithmes se fait toujours pour de grandes tailles de données.

L'analyse de performances d'un algorithme est indépendante du type de la machine sur laquelle il s'exécute. Elle ne fournit pas le nombre exact d'opérations mais un ordre de grandeur. En analyse d'algorithme, il n'y a pas de différences entre une solution qui fait N opérations et une autre qui fait N+300 opérations, car N et N+300 tendent vers la même limite quand N tend vers l'infini.

Exemple 2:

Soit le programme de tri par bulles (ci dessous) qui tri un tableau d'entiers par ordre croissant. Le programme doit son nom au fait que l'action principale de chaque exécution de la boucle centrale (3)-(6) est de faire remonter les éléments les plus petits du tableau vers le haut, comme les bulles d'un verre de boisson gazeifiée.

```
Algorithme TriBulle ; /* par ordre croissant */
Var I,J,temp :entier ;
Début
(1)  Pour I :=1 à n-1 faire
(2)      Pour J :=1 à n-1 faire
(3)          Si A[J]> A[J+1] alors
(4)              Temp :=A[J] ;
(5)              A[J] :=A[J+1] ;
(6)              A[J+1] :=temp ;
          Fsi ;
      Fait ;
  Fait ;
Fin ;
```

Les instructions de lignes (4) à (5) prennent chacune un temps de $O(1)$. Par la règle de la somme, la complexité conjointe de ce groupe d'instructions est $O(\max(1,1,1))=O(1)$.

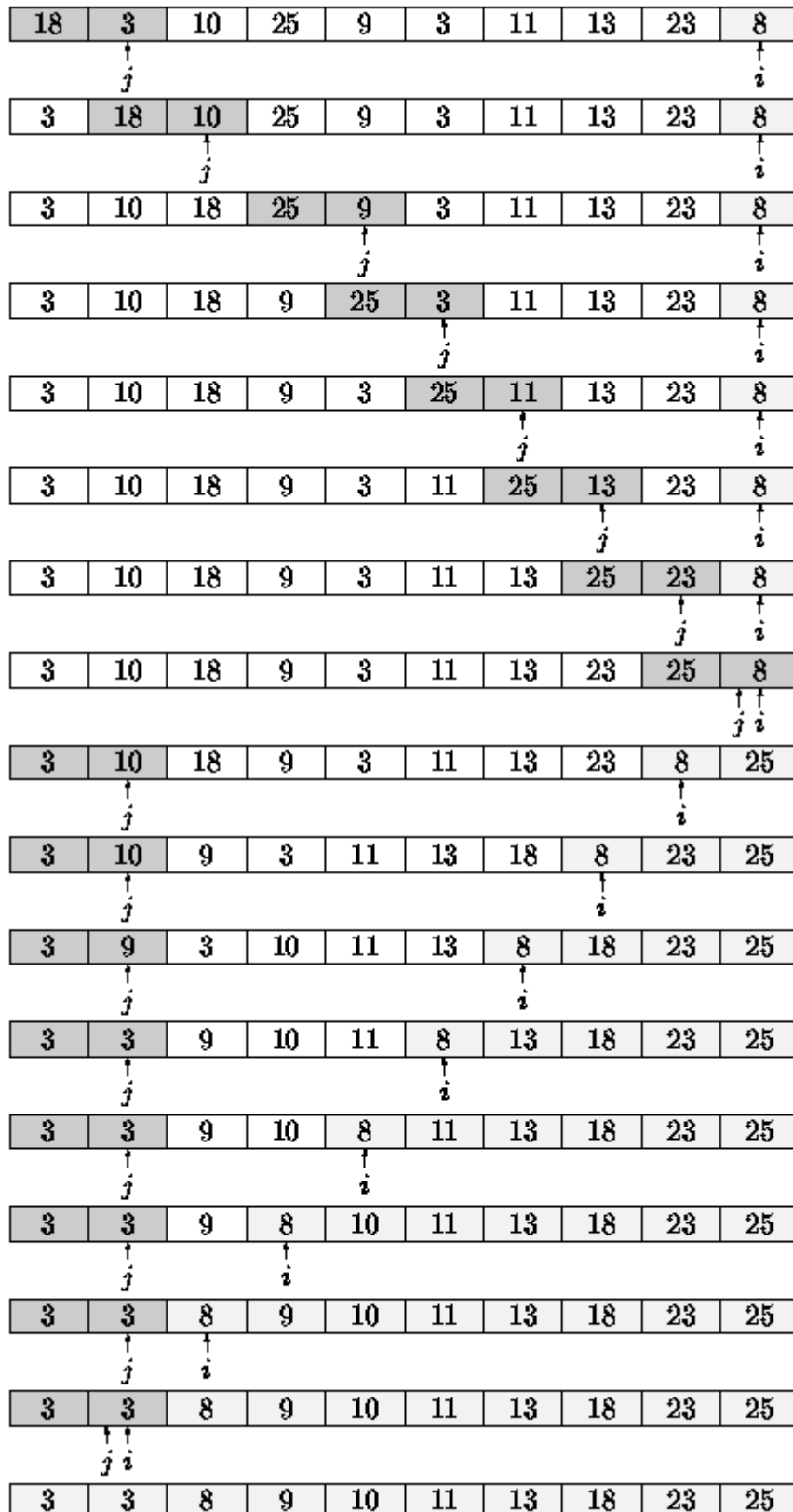
L'évaluation du **si** qui correspond à l'instruction (3) prend $O(1)$.

Les instructions du **si** sont toujours exécutées dans le pire des cas par conséquent les lignes (3) à (6) prennent $O(1)$.

En nous déplaçant vers l'extérieur, les instructions de (2) à (6) de la boucle **pour** sa complexité est donnée par la somme cumulée de la complexité de chaque itération qui est $O(1)$, comme le nombre d'itération est $(n-i)$ alors la complexité de ce groupe d'actions est $O((n-i)*1)=O(n-i)$.

Examinons maintenant la boucle externe qui contient toutes les instructions exécutables du programme. La ligne (1) est exécutée $n-1$ fois, et la complexité totale du programme est majorée par une certaine constante fois

$\sum_{i=1}^{n-1} (n-i) = n(n-1)/2 = n^2/2 - n/2$ ce qui donne un résultat de $O(n^2)$. Cette complexité est proportionnelle au carré du nombre de données qu'il trie.



Il est facile de compter le nombre d'opérations nécessaires. A chaque itération, on démarre à l'élément a_i et on le compare successivement à a_{i+1} , a_{i+2} , ..., a_N . On fait donc $N - i$ comparaisons. On commence avec $i = 1$ et on finit avec $i = N-1$. Donc on fait $(N-1) + (N-2) + \dots + 2 + 1 = N(N-1)/2$ comparaisons, et $N-1$ échanges. Le tri par sélection fait donc de l'ordre de N^2 comparaisons. Si $N = 100$, il y a 5000 comparaisons, soit 5 ms si on arrive à faire une comparaison et l'itération de la boucle « pour » en 1µs, ce qui est tout à fait possible sur une machine plutôt rapide actuellement. On écrira que le tri par sélection est en $O(N^2)$. Son temps est quadratique par rapport aux nombres d'éléments du tableau.

2.8 Types d'algorithmes

Un algorithme est soit itératif soit récursif. L'algorithme **itératif** est constitué de structures de contrôles : les boucles et le choix. L'analyse consiste à estimer le nombre de répétitions des itérations (§2.7).

Les algorithmes **récursifs** consistent à ramener la solution d'un problème de taille n à celle d'un nombre réduit de sous problèmes de même type que le problème initial mais de taille $m < n$. L'analyse d'un algorithme récursif consiste à estimer le temps d'exécution de tous les sous problèmes et leur composition pour résoudre le problème en entier. La combinaison des analyses de chaque sous problème fournit une **relation de récurrence** qui sera transformée en une équation qui relie le nombre d'opérations total à la taille du problème initial.

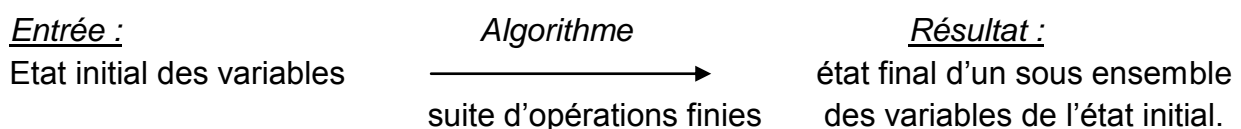
Exemple : Calcul de factorielle N (solution récursive)

$$n! = \begin{cases} 1 & \text{si } n=0, 1 \\ n*(n-1)! & \text{si } n>1 \end{cases} \Rightarrow T(n) = \begin{cases} 0 & \text{si } n=0, 1 \\ 1+T(n-1) \end{cases} \Rightarrow T(n) = \Theta(n)$$

***** **Avantages et inconvénients de la récursivité** *****

2.9. Validation d'un algorithme

Un algorithme est formé d'un ensemble fini d'opérations liées par une structure de contrôle (boucle, choix, ...) qui opèrent sur un ensemble fini de variables.



Un algorithme qui résout un problème P est valide si pour tout ensemble de valeurs de variables d'entrée, la suite des opérations exécutées est finie (condition de terminaison) et lorsque le programme termine, le sous-ensemble des variables de sortie contient le résultat recherché.

Donc prouver un algorithme consiste à :

- a. démontrer qu'il termine
- b. démontrer qu'il calcule bien ce qui est demandé (le résultat est valide)

Les questions qu'on se pose à propos d'un algorithme :

1. Prouver sa validité
 - Terminaison
 - Correction du résultat
2. Calculer sa complexité

2.9.1 Preuves de programmes

2.9.1.1 preuve par induction

Le terme induction appartient au vocabulaire des mathématiques. Le petit Robert y voit une *opération mentale qui consiste à remonter des faits à la loi, à des cas donnés*. Dans le cadre des mathématiques, l'induction est une méthode de preuve, permettant de montrer qu'une proposition est vraie pour n'importe quelle valeur d'un paramètre n . Pour ce faire, on commence par montrer que la proposition est vraie pour certaines valeurs *initiales* bien choisies de n (par exemple $n = 0$). C'est ce qu'on appelle le «cas de base1 ». Ensuite, on prouve que *si* la proposition est vraie pour certaines valeurs (dont les valeurs initiales), *alors* elle est aussi vraie pour d'autres. On appelle cette partie de la preuve le « cas inductif ». Le cas de base et le cas inductif permettent de construire une chaîne de raisonnement montrant que la proposition est vraie pour toute valeur.

Comme exemple de cas inductif, on montre que si la proposition est vraie pour $n = k$, avec $k \geq 0$, elle est aussi vraie pour $n = k+1$. Ainsi, nous obtenons la chaîne de raisonnements suivante : puisque la proposition est vraie pour $n = 0$ (ce qui est vrai, d'après le cas de base), elle est aussi vraie pour $n = 1$ (on utilise ici le cas inductif).

Maintenant que l'on sait que la proposition est vraie pour $n = 1$, on en déduit, grâce au cas inductif à nouveau que la proposition est vraie pour $n = 2$. Elle est donc aussi vraie pour $n = 3$, pour $n = 4$, etc. La proposition est donc vraie pour toute valeur entière positive de n .

En informatique, et plus particulièrement en algorithmique dès lors qu'il s'agit de raisonner formellement sur certaines propriétés d'algorithmes, les preuves par induction sont très souvent utilisées.

Preuve par induction :

Cas de base la proposition est vraie pour $n = 0$.

Cas inductif si la proposition est vraie pour $n = k$, où $k \geq 0$, alors elle l'est pour $n=k+1$.

2.9.1.2 Récurrence

Le terme « récurrence » recouvre lui aussi une certaine idée de répétition. Dans le cadre des mathématiques, on parle de *définition par récurrence* ou de *définition récursive*, quand on définit un concept par rapport à lui-même.

Exemple : on peut définir le calcul de la factorielle n ($n!$) par la solution classique :

$$n! = 1 \times 2 \times 3 \times \dots \times n-1 \times n$$

Algorithme1 :

Fonction fact(entier n) :entier ;

Debut

f, i :entier ;

$f \leftarrow 1$;

 pour $i \leftarrow 1$ à n faire

$f \leftarrow f * i$;

 finpour ;

 retourner(f) ;

Fin ;

Ou bien

$$n! = \begin{cases} 1 & \text{si } n=0, 1 \\ n*(n-1)! & \text{si } n>1 \end{cases}$$

La seconde définition est récursive, car dans le cas où $n > 1$, on définit la factorielle de n par rapport à une autre factorielle (en l'occurrence, celle de $n-1$).

Algorithme2 :

Fonction fact(entier n) :entier ;

Debut

 si $n=0$ ou $n=1$ retourner 1;

 sinon retourner $n*fact(n-1)$;

 finsi ;

Fin ;

La notion mathématique de récurrence a donné lieu à une technique de programmation, appelée *récursivité*, dans laquelle un algorithme fait appel à lui-même pour résoudre un problème.

La définition récursive de la factorielle suggère immédiatement l'*Algorithme1*, la définition classique, par contre, suggère plutôt une implémentation itérative l'*Algorithme2*.

Pour prouver l'*Algorithme1* on utilisera la ***preuve par induction*** et l'*Algorithme2* on utilisera la ***preuve par récurrence***.

Prouver un programme récursif, c'est :

- prouver que le résultat produit dans le cas d'arrêt est correct
- prouver que les résultats produits dans les cas d'appels récursifs, *sous hypothèse* que ces appels récursifs soient corrects et produisent bien ce qu'il est attendu d'eux, sont corrects
- prouver la convergence, c'est-à-dire que toute séquence d'appels récursifs conduit toujours à une situation d'arrêt.

2.9.2 Validation d'un algorithme

Montrons comment les techniques de preuves développées dans la section précédente permettent de raisonner sur des programmes itératifs. Ce lien est assez naturel, car pour prévoir l'effet de k itérations d'une boucle donnée, il faut en général connaître l'effet de $k-1$ itérations. En effet, la $k^{\text{ème}}$ itération s'exécute après les $k-1$ premières itérations, et son effet dépend donc des valeurs des variables après $k-1$ itérations.

On retrouve là l'esprit des preuves par induction : pour prouver qu'une certaine proposition est vraie après un nombre arbitraire d'itérations d'une certaine boucle, on montrera d'abord que la proposition est vraie avant la première itération ; puis on montrera que si la proposition est vraie pour $k-1$ itérations, elle l'est aussi pour k itérations.

Exemple : *Ecrire un algorithme qui calcul la somme des éléments d'un tableau A de n entiers ($n>0$)*

- 1) Ecrire un algorithme itératif qui calcule la somme des éléments de A et prouver sa validité
- 2) Déterminer sa complexité
- 3) Ecrire un algorithme récursif pour le même problème et prouvez-le
- 4) Déterminer sa complexité

Solution :

1) L'algorithme itératif

```
fonction somme (A : tableau ; n : entier) : entier ;  
debut  
  s, i : entier;  
  S ← 0 ;  
  pour i ← 1 à n faire  
    s ← s + A[i] ;  
  finpour;  
  retourner s;  
fin;
```

Pour prouver la validité de cet algorithme on démontre sa terminaison et la validité du résultat

a. La terminaison :

La preuve est triviale car : $n > 0$ par hypothèse ; $i = 1$ par initialisation et est incrémenté de 1 à chaque tour de boucle ; donc i atteindra la valeur d'arrêt n après $n-1$ itérations, aussi le corps de la boucle ne contient que l'addition et l'affectation qui s'exécutent chacune en un temps fini donc l'algorithme se termine.

b. La validité :

On la démontre par récurrence (sur la taille des données). On montre qu'à chaque itération le résultat est celui recherché et ceci en formalisant l'expression du résultat d'une itération.

On considère la propriété suivante :

« A la fin de l'itération i , la variable S contient la somme des i premiers éléments du tableau A »

Cette propriété est dite aussi : « L'invariant de la boucle »

Définition d'un invariant :

Un **invariant** est une propriété logique sur les valeurs des variables qui caractérise tout ou partie de l'état interne du programme et qui doit être « toujours » vraie. Dans le cas d'un programme récursif, cet invariant est la relation de récurrence ; dans le cas d'un programme itératif, cet invariant est l'invariant de boucle.

Preuve par récurrence de la propriété :

On note S_i cette somme. On a alors :

c. $S_0 = 0$

d. Pour $i=1$ $S_1 = S_0 + A[1]$

$S_1 = A[1]$ donc la propriété est vraie pour $i=1$

e. On suppose que la propriété est vraie à l'itération i , c'est-à-dire :

$$S_i = \sum_{j=1}^i A[j] \text{ vraie et montrons qu'elle est vraie à l'itération } i+1$$

« A la fin de l'itération $i+1$ la variable S contient ce qu'elle contenait à l'étape i , plus $A[i+1]$ »

$$S_{i+1} = S_i + A[i+1]$$

$$= \sum_{j=1}^i A[j] + A[i+1]$$

$$S_{i+1} = \sum_{j=1}^{i+1} A[j] \text{ donc la propriété est vraie à } i+1$$

L'algorithme se termine à la fin de l'itération n et on aura donc calculé : $S_n = \sum_{j=1}^n A[j]$

L'algorithme est donc valide.

c. La complexité

On compte le nombre d'opérations dans la boucle : addition + affectation

= 1 opération

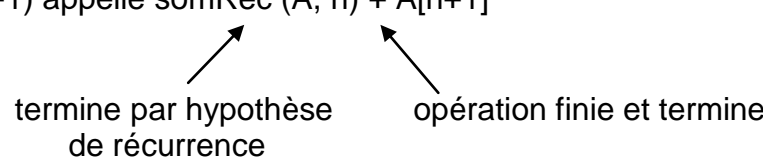
$T(n) = n \Rightarrow T(n) = O(n)$ et plus précisément $T(n) = \Theta(n)$

2) L'algorithme récursif

```
fonction somRec (A : tableau ; n : entier) : entier ;  
debut  
    si  $n \leq 0$  retourner 0  
    sinon retourner (A[n] + somRec (A, n-1));  
finsi ;  
fin;
```

a. La terminaison

Par récurrence sur i :

- Si $i=0$, cas évident, se termine sans appel récursif.
- On suppose qu'à l'étape n , l'algorithme termine donc $\text{somRec}(A, n)$ termine et évaluons $\text{somRec}(A, n+1)$
 $\text{somRec}(A, n+1)$ appelle $\text{somRec}(A, n) + A[n+1]$


termine par hypothèse de récurrence opération finie et termine

donc $\text{somRec}(A, n+1)$ termine pour tout n

b. La validité

On note S_n la valeur retournée par $\text{somRec}(A, n)$. La fonction reproduit la relation de récurrence suivante :

$$\begin{cases} S_0=0 \\ S_n= S_{n-1} + A[n] \text{ si } n>0 \end{cases}$$

On montre comme dans le cas itératif $S_n = \sum_{j=1}^n A[j]$

Et donc que l'algorithme calcule bien la somme des éléments du tableau A .

c. La complexité

On compte le nombre d'addition :

$$T(n) = \begin{cases} 0 & \text{si } n=0 \\ T(n-1) + 1 & \text{si } n>0 \end{cases}$$

Qui a pour solution $T(n) = n$. Et on en déduit que $T(n)=\Theta(n)$

Méthode générale :

Pour résoudre une équation de récurrence, on remplace le terme $T(k)$ par le membre droit de l'équation k fois pour obtenir une formule qui ne contient plus « T » à droite et on intègre à la fin les cas particuliers dans la formule.

Exemple : Calcul de $n!$

fonction fact (n :entier)

debut

1. Si ($n \leq 1$) alors retourner 1

2. Sinon retourner($n \cdot \text{fact}(n-1)$) ;

finsi ;

fin ;

Soit $T(n)$ le temps d'exécution de fact(n)

- La ligne 1 : s'exécute en $O(1) = \text{constante} = d$
- La ligne 2 : s'exécute en $O(1) + T(n-1) = \text{constante} + T(n-1)$
 $= c + T(n-1)$

Donc il existe 2 constantes c et d telles que :

$$T(n) = \begin{cases} d & \text{si } n \leq 1 \\ c + T(n-1) & \text{si } n > 1 \end{cases} \quad (1)$$

Soit $n > 2$, appliquons la formule (1)

Pour $n-2$:

$$\left. \begin{array}{l} T(n) = c + T(n-1) \\ T(n-1) = c + T(n-2) \end{array} \right\} \Rightarrow T(n) = 2 \cdot c + T(n-2)$$

$$T(n-2) = c + T(n-3) \Rightarrow T(n) = 3 \cdot c + T(n-3)$$

$$\text{Donc } T(i) = i \cdot c + T(n-i) \quad n > i$$

$$\text{Pour } i=n-1 \Rightarrow T(n) = (n-1) \cdot c + T(1) \Rightarrow T(n) = (n-1) \cdot c + d = O(n)$$

$$\text{Donc } T(n) = O(n)$$