Algorithmique Avancée et Complexité

PLAN

1. Introduction
2. Les bases de l'analyse d'un algorithme
3. Calcul de la complexité d'un algorithme
4. Preuves de programmes
5. Méthode de résolution des équations de récurrence
6. Les structures de données
7. Les classes de problèmes

1. Introduction

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

- existe t-il un algorithme pour résoudre le problème en question ?

- 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.

Calculabilité

La théorie de la calculabilité est une branche de la logique mathématique et de l'informatique théorique elle est née à la fin du 19ème 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.

.

- En calculabilité on cherche à définir les problèmes dont les solutions sont calculables par un algorithme et ceux qui ne le sont pas.

- Une bonne appréhension de ce qui est calculable et de ce qui ne l'est pas permet de voir les limites des problèmes que peuvent résoudre les ordinateurs.

Au début du 20ème siècle, ***David Hilbert*** avait pour projet de rechercher les principes de base des mathématiques.

Il se propose de :

- Formaliser toutes les mathématiques.
- Trouver les axiomes et les règles de déductions qui permettraient de démontrer toutes les vérités mathématiques.
- Mécaniser (automatiser) le raisonnement mathématique.

Mais en 1931, le mathématicien ***Kurt Gödel*** publie son théorème d'incomplétude qui démontre les limites du projet de **Hilbert**:

"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.

Ainsi :

Tout n'est pas démontrable.

Conséquences du théorème d'incomplétude sur l'informatique

- Tout problème ne peut être résolu par un algorithme.

- Impossible de concevoir un programme capable de vérifier tous les programmes.

Décidabilité

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.

Exemple

Équations diophantiennes

Déterminer si une équation de la forme

P=0, où P est un polynôme à coefficients

entiers, possède des **solutions entières**

– Exemples : $x^2 + y^2 - 1 = 0$,

$x^2 - 991y^2 - 1 = 0$

etc…

Ce problème n’est pas « décidable »

Démontré en 1970 par Yuri Matijasevic

Qu'est-ce que cela signifie?

Il n'existe aucun algorithme qui indique, pour chaque équation diophantienne, si elle a ou non des solutions entières

Cas de l'informatique

- Déterminer si un programme P, pris au hasard, calcule une fonction donnée non nulle *f(n)* (n entier)
- Déterminer si deux programmes calculent la même chose (sont équivalents)
- Déterminer si un programme quelconque, sur une donnée représentée par un entier n, ne va pas boucler indéfiniment

Ce sont des problèmes **« indécidables »**

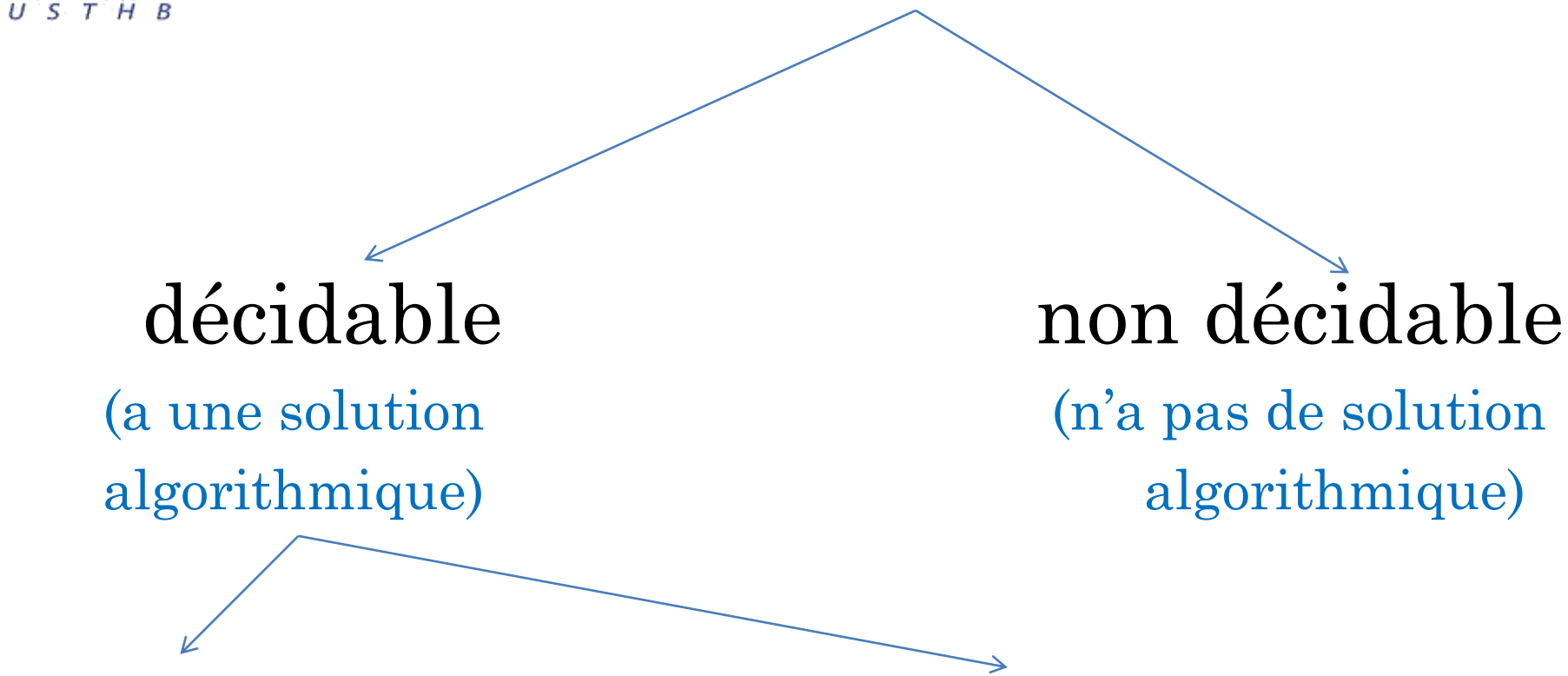
Décidabilité

Pour les problèmes décidables, il n'était plus suffisant de savoir qu'une solution existe, mais il devenait impératif de savoir que la solution soit réalisable, "exécutable" de manière "raisonnable".

C'est le domaine de la

Complexité des algorithmes.

Problème


décidable
(a une solution algorithmique)
non décidable
(n'a pas de solution algorithmique)
exécutable
de manière raisonnable
non exécutable

Complexité

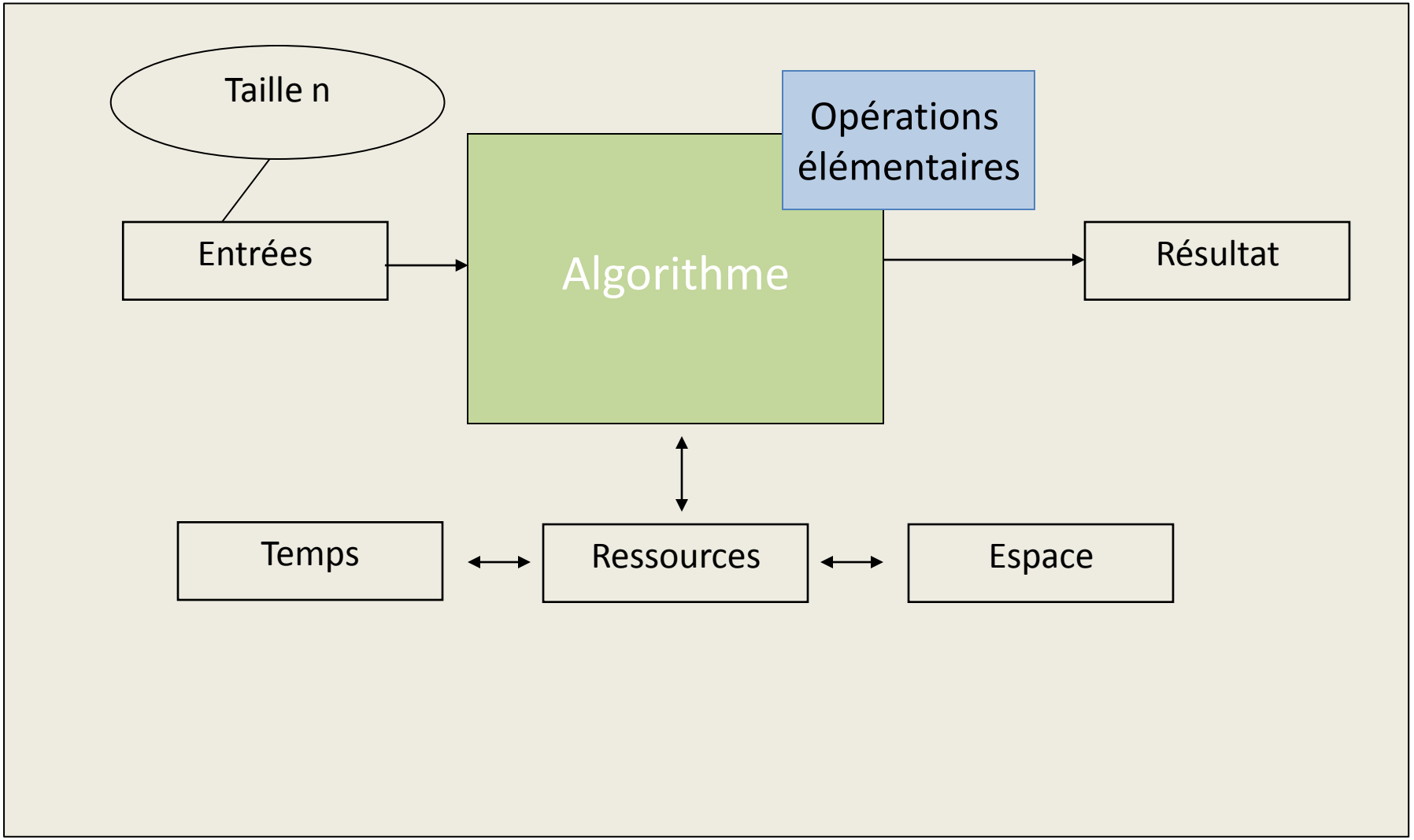
- La complexité c'est la maîtrise de l'algorithmique.
- Elle vise à déterminer quels sont les temps de calcul et espace mémoire requis pour obtenir un résultat donné.

Complexité

- 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 temps de calculs très lents,
- des débordements mémoire conséquents, qui font "planter" la machine.

Evaluation d'un algorithme



Problème :

Un problème est une **question générique** qui a les propriétés suivantes :

- Elle s'applique à un ensemble d'éléments et toute question posée pour chaque élément admet une réponse.
- 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 :

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

Instance :

Une instance de problème est la question générique appliquée à un élément. Un problème contient des paramètres ou des variables libres et lorsque l'on attribue une valeur à ces variables libres on obtient une instance du problème.

Exemples :

- *L'entier 15 est-il pair ou impair ?*
- *Déterminer le maximum de { 1, -5, 14, 9} ;*
- *Trier la suite d'entiers précédente, etc.*

Problème de décision

Un problème de décision est un problème pour lequel la réponse est dans l'ensemble

{ OUI - NON }

Exemples :

-Un nombre N est-il un multiple de 4?

-Un voyageur de commerce désire faire sa tournée, existe-t-il une tournée de moins de 50km?

Problème de décision

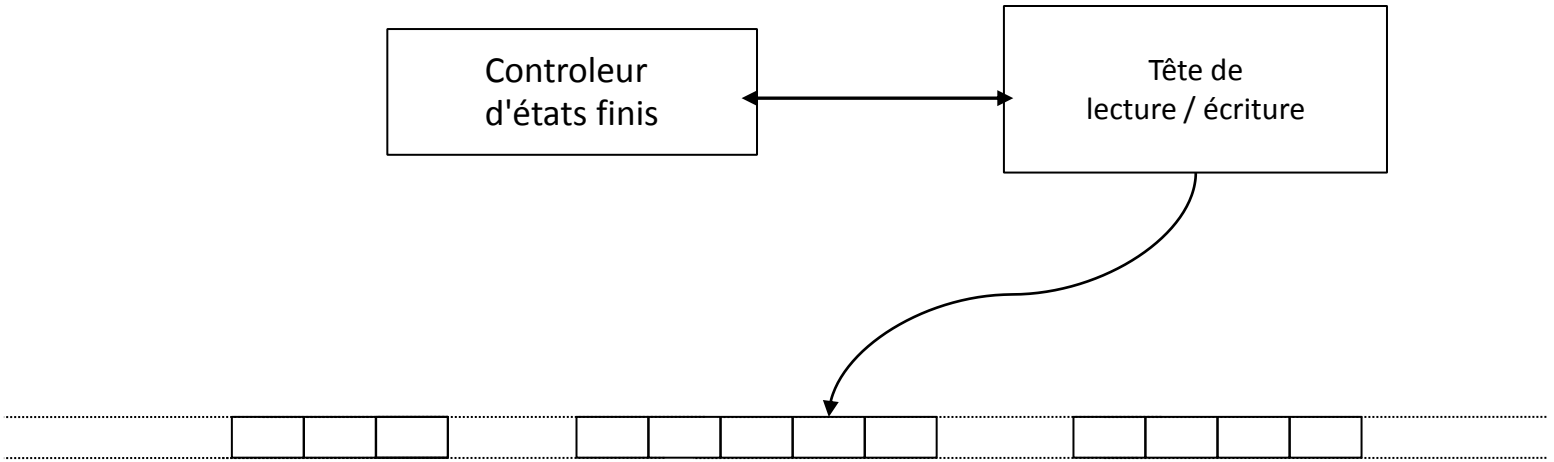
Problème de calcul qui consiste à déterminer si une certaine entrée w fait partie d'un langage L. En pratique on ne fera pas la différence entre un langage et un problème de décision.

Problème de décision & langage accepté par une machine de Turing

La machine de Turing est constitué de :

- **Un ruban infini** découpé en cases avec un symbole dans chaque case, une tête de lecture et d'écriture positionnée sur une case du ruban et une unité centrale qui contient le programme et un état.
- Chaque instruction est un quintuplet ***<Q,A,A',Q',Dir>*** :
- Si l'état de la machine est Q et si A est le symbole lu par la tête de lecture alors remplacer A par A', déplacer la tête dans la direction Dir (Droite ou Gauche) et passer de l'état Q à l'état Q'.

Machine de Turing



$\sum$ l 'ensemble des symboles pouvant être écrit par l 'utilisateur sur le ruban

$\sum^*$ l 'ensemble des mots du langage sur l 'alphabet

$L_M = \{ x \in \sum^* : M \text{ accepte } x \}$

Résoudre le problème revient à savoir si $x \in L_M$

Solution :

- C'est une procédure effective 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.

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.

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.

<u>Exemple :</u> 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.

Algorithme:

Voici deux définitions :

- Un algorithme est une procédure de calcul bien définie qui prend en entrée un ensemble de valeurs et produit en sortie un ensemble de valeurs. (Cormen, Leiserson, Rivert)
- Un algorithme est une spécification d'un schéma de calcul sous forme d'une suite finie d'opérations élémentaires obéissant à un enchaînement déterminé.

  (Al Khowarizmi, Bagdad IXè siècle. *Encyclopedia Universalis)*

Exemple : Enoncé du problème : "trier un ensemble de n ≥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** *(Entrées)* tel que le $i^{ème}$ entier soit stocké à la $i^{ème}$ position : t[i]; 0 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[min];

permuter t[i] et t[min];

}

Sorties : le même tableau trié par ordre croissant

Pour comparer des solutions, plusieurs points peuvent être pris en considération

- Exactitude des programmes
- Simplicité des programmes
- Convergence et stabilité des programmes
- Efficacité des programmes

L’analyse d’un algorithme

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.

Les phases à suivre pour analyser un algorithme

Phase1 : soit p un problème, il s'agit de l'énoncé du problème

Phase2: soit M une méthode, pour résoudre un problème p et élaboration de l'algorithme:

- Entrées
- Sorties
- Étapes de résolution

C'est donc la description de M dans un langage algorithmique

Les phases à suivre pour analyser un algorithme

Phase3 : vérification ou validation, c'est de montrer que l'algorithme se termine et fait bien ce que l'on attend de lui.

Phase4: mesurer l'efficacité de l'algorithme indépendamment de l'environnement (machine, système, compilateur, …)

Les phases à suivre pour analyser un algorithme

On va calculer la complexité théorique de l'algorithme en évaluant le nombre d'opérations élémentaires (affectation, comparaison, boucle, …) en fonction de la taille des données et de la nature des données.

Notations: *n: taille des données*

T(n): nombre d'opérations élémentaires

Phase5: mise en œuvre → programmation

Les paramètres de l'analyse

- *La taille des entrées*
- *La distribution des données*
- *La structure de données*

Les types de complexité

- *complexité du meilleur cas*
- *complexité en moyenne*
- *complexité du cas pire*

Complexité temporelle

Deux approches:

- approche théorique
- approche pratique

Complexité temporelle

Approche théorique

- Comportement général de l'algorithme
- Les détails ne comptent pas
- Le comportement de l'algorithme est de l'ordre d'une certaine fonction

Complexité temporelle

Approche pratique

- Il s'agit d'obtenir une évaluation beaucoup plus fine des algorithmes
- Toutes les instructions sont prises en compte « décortiquées »
- Fournit des indications fines
- Détection des zones sensibles
- Aide à l'optimisation de programmes

Complexité théorique ou Asymptotique

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.

Complexité Asymptotique

Plus simplement c'est le comportement de la complexité d'un algorithme quand la taille des données devient grande.

Les notations Landau

- 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 ~ (de l'ordre de ; équivalent à)

Les notations Landau

- **Notation grand O :**

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

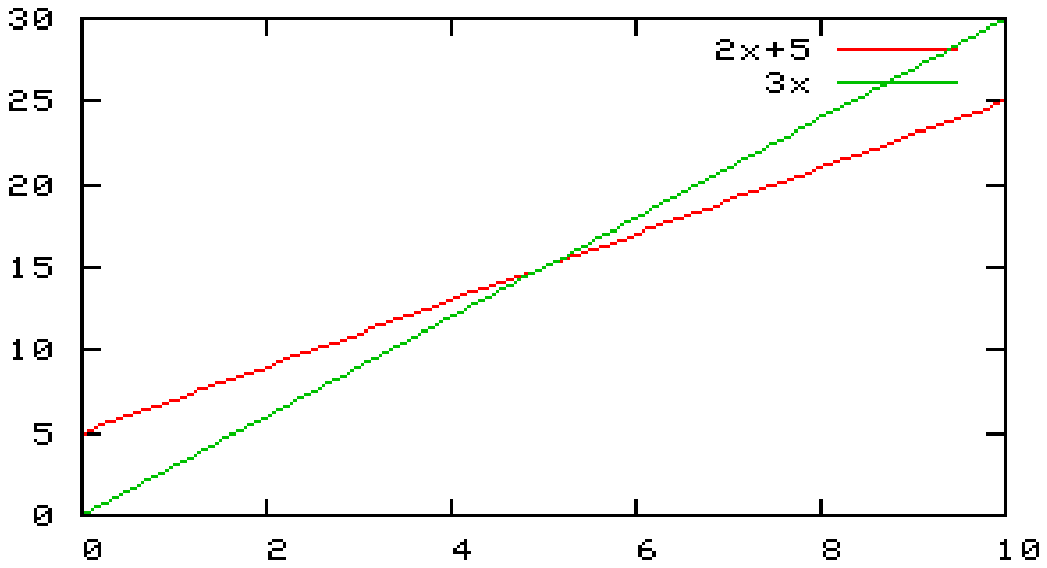
- *On dit que g(n) est une* ***borne supérieure*** *asymptotique pour f(n), on note abusivement* $f(n) = O(g(n))$.

Notation grand O

Exemple : $f(n) = 2n+5 = O(n)$ car $2n+5 \leq 3n$

$\forall n \geq 5$ *d'où* $c=3$ *et* $n_0=5$

Le graphe ci-dessous illustre l'exemple :



Les notations Landau

- **Notation Ω**

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

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

Notation Ω

Exemple : f(n)=2n+5= Ω (n)

car $2n+5 \geq 2n$

$\forall\, n \geq 0$ *d'où* $c=2$ *et* $n_0=0$

Les notations Landau

- **Notation Θ**

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

Si f(n) ∈ Θ(g(n)) On dit que g(n) est ***une borne asymptotique*** *pour f(n), on note abusivement f(n) = Θ(g(n))*

Notation Θ

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.

Propriétés

- Réflexivité : $f(n) = O(f(n))$.
- Transitivité : *si* $f(n) = O(g(n))$ *et* $g(n)=O(h(n))$ *alors* $f(n)=O(h(n))$.
- Somme : *si* $f(n) = O(h(n))$ *et* $g(n) = O(h(n))$ *alors* $f(n)+g(n)=O(h(n))$.
- Produit : *si* $f(n) = O(F(n))$ *et* $g(n) = O(G(n))$ *alors* $f(n)xg(n) = O(F(n)x\ G(n))$;

  en particulier $cxf(n) = O(F(n))$ pour toute constante c, car toute fonction constante est O(1).

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

Propriétés

- On peut aisément déduire les propriétés suivantes :
- Si $p(n)$ est un polynôme de degré k alors $p(n)=\Theta(n^k)$.
- $Log_b\, n = \Theta(log_b\, n)$. 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.
- $(n+1)^k = O(n^k)$.

Vocabulaire

- *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 u*n 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, …

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.

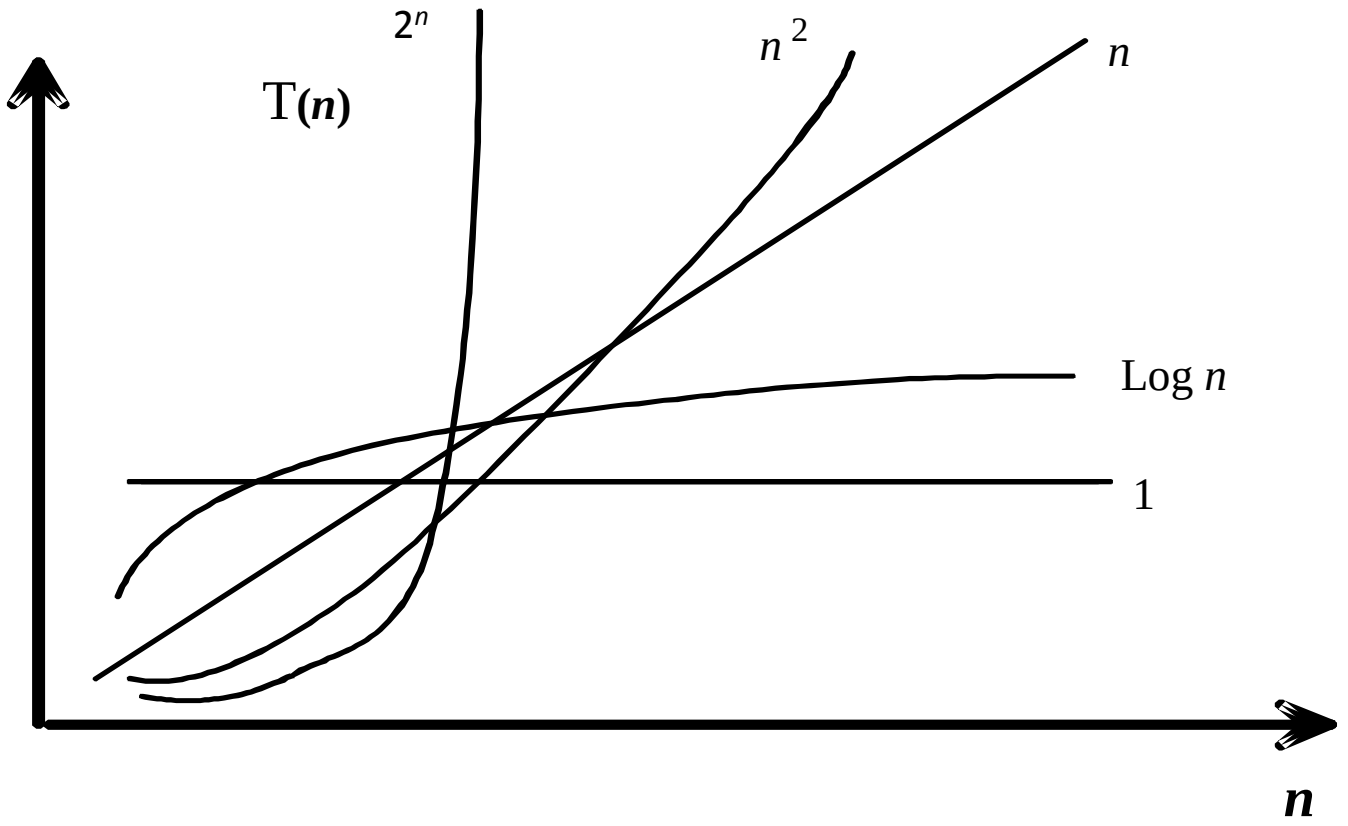
Exemple : si $T(n)=2^n$

pour n=60 $T(n)=1,15 \times 10^{12}$ secondes

$1,15 \times 10^{12}$ secondes 36.558 ans

et si $T(n)=n !$ on a $n ! >> 2^n$

Ordres de grandeur (exemples)
1, log n, n, n.log n, n^2, n^3, 2^n



Deux types d'algorithme:

- Itératif
- récursif

Estimation du coût d'un algorithme itératif

C'est le nombre d'opérations élémentaires, affectations, comparaisons, opérations arithmétiques, effectuées par l'algorithme.

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.

Estimation du coût d'un algorithme itératif

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.

Estimation du coût d'un algorithme itératif

Somme des coûts:

traitement1 = coût_1

traitement2 = coût_2

…

traitementi = coût_i

…

Coût_total=coût_1+coût_2+…coût_i+…

Estimation du coût d'un algorithme itératif

```
f(…)
{  si  (…)  alors g(…) ;
            Sinon h(…) ;
   finsi ;
}
```

coût de f(…)=max(coût de g(…), coût de h(…))

Estimation du coût d'un algorithme itératif

```
f(…,  n :entier)
{   pour i :=1 à n faire
        g(…) ;
}
```

Coût de f(…,n)= n x coût de g(…)

Si g(…) dépend de la valeur de i alors le

$$coût\ de\ f(\ldots,n) = \sum_{i=1}^{n} coût(g(\ldots,i))$$

Estimation du coût d'un algorithme itératif

f(..., n :entier)

{ tantque <condition> faire

traitement g(...) ;

}

Coût de f(...,n)= $\sum_{i=1}^{k} coût(g(n))$

Estimation du coût d'un algorithme
Règles de la notation O

Facteurs constants :

- *La première règle à retenir est que :*

tout facteur constant est simplifié à 1.

- *La deuxième règle est que :*

les constantes multiplicatives sont omises.

Estimation du coût d'un algorithme
Règles de la notation O

- *Règle de la somme* :
  $O(f(n)+g(n))=O(max(f(n),g(n)))$

- *Règle de la multiplication*

$$O(f(n)xg(n)) = \begin{cases} O((f(n)xf(n))) \ \ si\ f(n)>g(n) \\ \\ O((g(n)xg(n))) \ \ si\ g(n)>f(n) \end{cases}$$

Estimation du coût d'un algorithme

Exemple:

Calcul de e^x à l'aide d'un développement limité de n termes:

$$S = e^x \approx 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!}$$

- Quel est l'ordre de grandeur du temps d'exécution?

Estimation du coût d'un algorithme

1ère solution:

```
s=1;
Pour i=1 à n faire
        p=1;
        pour j=1 à i faire
            p=p*x/j;
         finpour
s=s+p
finpour;
```

Estimation du coût d'un algorithme

La boucle intérieure est exécutée:

$$1+2+\ldots+n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Cette solution a une complexité de l'ordre de :

$$O(n^2)$$

Estimation du coût d'un algorithme

2ème solution:

$$\frac{x^i}{i!} = \frac{x}{i} * \frac{x^{i-1}}{(i-1)!}$$

```
S=1; p=1;
pour i=1 à n faire
    p=p*x/i;
    s=s+p;
fait;
```

Estimation du coût d'un algorithme

Cette fois le corps de la boucle n'est exécutée que n fois donc la complexité est de l'ordre de:

$$O(n)$$

Estimation du coût d'un algorithme

Exercices:

Montrer que :

1) $f(n) = 5n^2 - 6n = \theta(n^2)$

2) $f(n) = 6n^3 \neq \theta(n^2)$

3) $n^2 \in O(10^{-5} * n^3)$

Validation d'un algorithme

Entrée : Etat initial des variables

Algorithme: suite d'opérations finies des variables de l'état initial.

Résultat : état final d'un sous ensemble

Validation d'un algorithme

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

- Prouver sa validité
- Terminaison
- Correction du résultat
- Calculer sa complexité

Preuves de programmes

Preuve par induction:

L'induction est une méthode de preuve, permettant de montrer qu'une proposition est vraie pour n'importe quelle valeur d'un paramètre n.

Preuves de programmes

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 $n=k+1$.

Preuves de programmes

Exemple :

Montrer que = $2^{n+1} - 1$

n=0 = $2^1 - 1$

$2^0 = 2^1 - 1$ $2^0 = 1$ *et* $2^1 - 1 = 1$ *la proposition est vraie pour n=0*

n=k $= 2^{k+1} - 1$ *supposée vraie*

Montrons que pour n=k+1 la proposition est vraie c'est-à-dire $= 2^{k+1} - 1$

Preuves de programmes

$$\sum_{i=0}^{k+1} 2^i = + 2^{k+1}$$

$$= 2^{k+1} - 1 + 2^{k+1}$$

$$= 2x2^{k+1} - 1$$

$$= 2^{k+2} - 1$$ *ce qu'il fallait démontrer donc l'égalité est vraie.*

Preuves de programmes

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. Un algorithme récursif fait appel à lui-même pour résoudre un problème.

Preuves de programmes

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

n!= 1x 2x 3x…x n-1 x n

Preuves de programmes

Algorithme1 :

```
Fonction fact(entier n) :entier ;
Debut
  f, i :entier ;
f←1 ;
pour i←1 à  n faire
   f←f*i ;
finpour ;
retourner(f) ;
Fin ;
```

Preuves de programmes

Ou bien

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

Preuves de programmes

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.

Preuves de programmes

Validation d'un algorithme:

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.

Preuves de programmes

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

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

Preuves de programmes

Exemple: calcul de la somme de n nombres

Solution :

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

Preuves de programmes

Exemple: calcul de la somme de n nombres

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.

Preuves de programmes

Exemple: calcul de la somme de n nombres

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 »

Preuves de programmes

Exemple: calcul de la somme de n nombres

La validité :

Plus formellement : $S_i = \sum_{k=1}^{i} A[i]$

Cette propriété est dite aussi :

« L'invariant de la boucle »

Preuves de programmes

Exemple: calcul de la somme de n nombres

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.

Preuves de programmes

Exemple: calcul de la somme de n nombres

Preuve par récurrence de la propriété :

On note S_i cette somme. On a alors :

$S_0 = 0$

Pour i=1 : on a $S_1 = S_0 + A[1]$

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

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

$$S_i = \sum_{k=1}^{i} A[k] \text{ vraie}$$

et montrons qu'elle est vraie à l'itération i+1

Preuves de programmes

Exemple: calcul de la somme de n nombres

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

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

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

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

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

L'algorithme est donc valide.

Preuves de programmes

Exemple: calcul de la somme de n nombres

La complexité

On compte le nombre d'opérations dans la boucle :

addition + affectation

= 1 opération

T(n) = n

T(n) = O(n) et plus précisément T(n) = Θ(n)

Preuves de programmes

Exemple: calcul de la somme de n nombres

L'algorithme récursif

fonction somRec *(A : tableau ; n :entier) :entier ;*

debut

si n=0 retourner 0

sinon retourner (A[n] + somRec (A, n-1));

finsi ;

fin;

Preuves de programmes

Exemple: calcul de la somme de n nombres

La terminaison

Par récurrence sur n :

Si n=0, cas évident, se termine sans appel récursif.

On suppose qu'à l'étape n, l'algorithme termine donc somRec (A, n) termine et évaluons somRec (A, n+1)

Preuves de programmes

Exemple: calcul de la somme de n nombres

La terminaison

somRec (A, n+1) appelle somRec (A, n) + A[n+1]

termine par hypothèse de récurrence

addition opération finie et termine

donc somRec (A, n+1) termine $\forall$ la valeur du paramètre n

Preuves de programmes

Exemple: calcul de la somme de n nombres

La validité

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

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

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.

Preuves de programmes

Exemple: calcul de la somme de n nombres

La complexité

On compte le nombre d'addition :

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

Qui a pour solution T(n) = n.

Et on en déduit que **$T(n)=\Theta(n)$**

Preuves de programmes

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.

Preuves de programmes

Exemple : *Calcul de n !*

```
fonction fact (n :entier)
debut
Si (n<=1)  alors retourner 1
Sinon retourner(n*fact(n-1) ;
 finsi ;
fin ;
```

Preuves de programmes

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

La ligne 1 : s'exécute en O(1)=constante = d

La ligne 2 : s'exécute en O(1) + T(n-1) = constante +T(n-1)

= c+T(n-1)

Donc il existe 2 constantes c et d telles que :

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

Soit n>2, appliquons la formule (1)

Preuves de programmes

Pour n-2 :

$T(n) = c + T(n-1)$

$T(n-1) = c + T(n-2)$

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

$T(n-2) = c + T(n-3)$ $T(n) = 3*c + T(n-3)$

Donc $T(i) = i*c + T(n-i)$ $n>i$

Pour $i=n-1$ $T(n) = (n-1)*c + T(1)$ $T(n) = (n-1)*c + d = O(n)$

Donc $T(n)=O(n)$