Algorithmique Avancé 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

Références bibliographiques

1. Cormen, Leiserson, Rivest Stein, *Algorithmique, cours exercices et problèmes*, 3ème édition, Dunod, 2010.

2. Beauquier, Berstel, Chrétienne, *Éléments d'algorithmique,* édition Masson 2005.

Introduction

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

1. Existe t-il un algorithme pour résoudre le problème en question ?

2. 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.

Notion de 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.

- 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 que:

"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: Exemples de problèmes « indécidables »

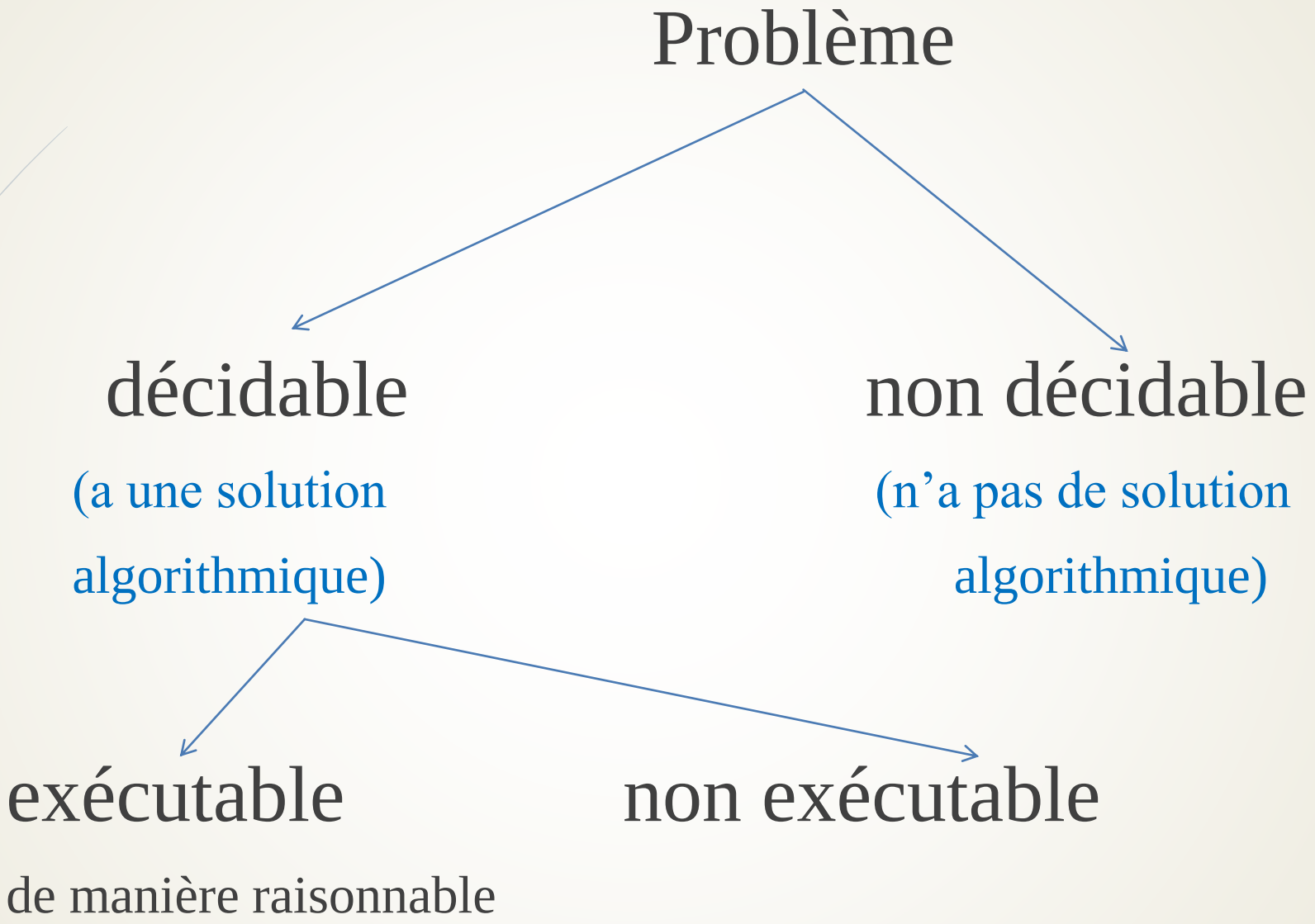
- Déterminer si un programme P, pris au hasard, calcule une fonction donnée non nulle *f(n)* (n entier)
- Déterminer si deux programmes calculent la même chose (sont équivalents)
- Déterminer si un programme quelconque, sur une donnée représentée par un entier n, ne va pas boucler indéfiniment

Complexité

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

Quelques définitions

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

Plus précisément:

- 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.

Problème

Un problème est une **question générique** auquel un ordinateur pourrait répondre et qui a les propriétés suivantes :

- 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 (exemple).
- Un problème contient des paramètres ou des variables et lorsque l'on attribue une valeur à ces variables on obtient une instance du problème.
- Un problème se compose donc d'une ou plusieurs questions et d'un ensemble d'instances.

Exemples :

- *L'entier 15 est-il pair ou impair ?*
- *Déterminer le maximum de { 1, -5, 14, 9} ;*
- *Trier la suite d'entiers { 1, -5, 14, 9, 3, -7},*
- *etc.*

Solution

- Une solution est une réponse à un problème, à une question : *Trouver la solution d'une énigme.*
- Une solution pour un problème est un algorithme qui répond correctement à la question pour chaque instance possible.
- Une solution à un problème pouvant être exécuté par un ordinateur est une procédure effective.

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 de savoir si une boucle est infinie ou non.

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.

Algorithme

- 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]; $0 \leq i < n$.

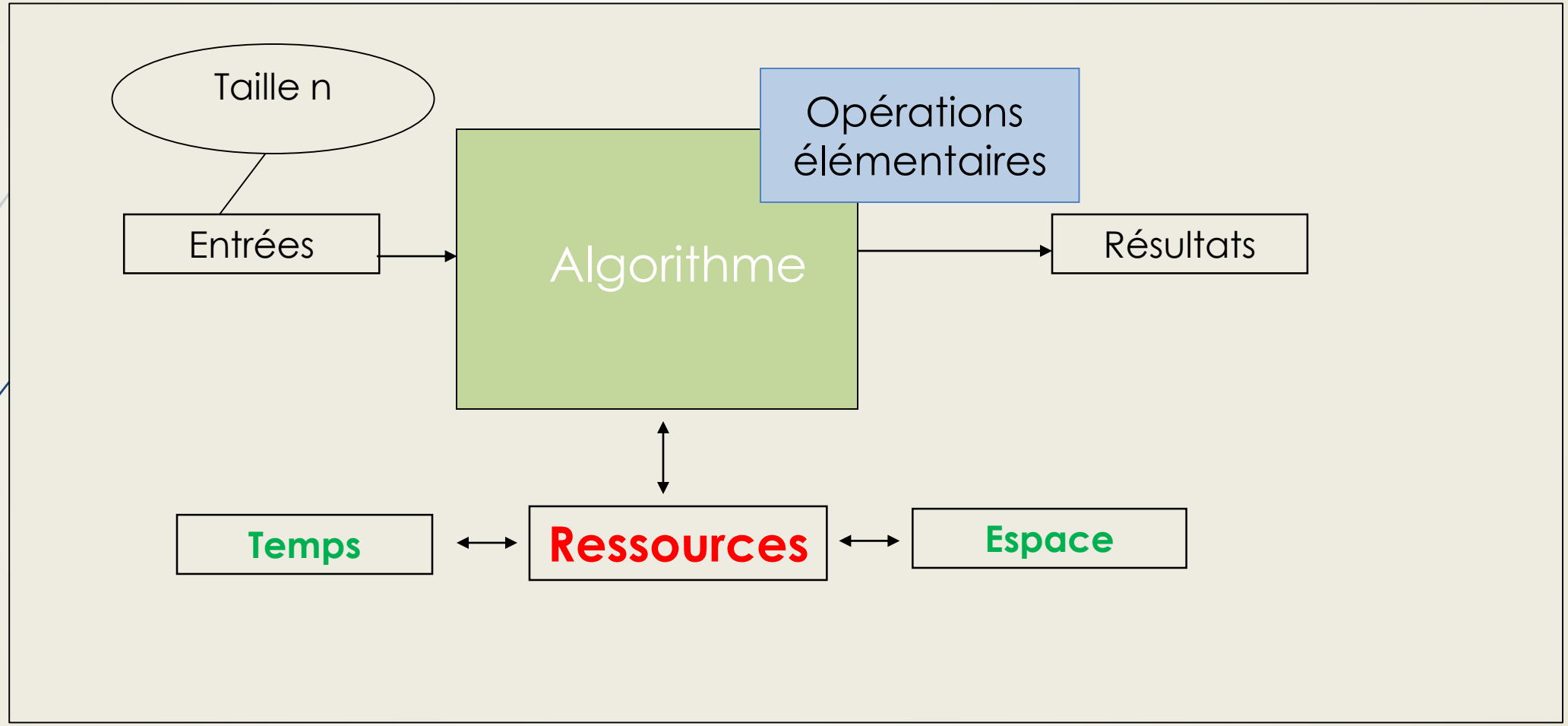
L'algorithme suivant est la solution d'un tri par sélection.

```
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[min];
        permuter t[i] et t[min];
}
```

Sorties : le même tableau trié par ordre croissant

Les bases de l’analyse d’un algorithme

Analyse d'un algorithme



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.

Complexité temporelle et spatiale

Evaluation du temps d'exécution de l'algorithme →Complexité en temps

Evaluation de l'espace mémoire occupé par l'exécution de l'algorithme →Complexité en espace

Les phases à suivre pour analyser un algorithme

Phase 1 : soit P un problème, il s'agit de l'énoncé du problème

Phase 2 : soit **M** une méthode, pour résoudre le 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

Phase 3 : vérification ou validation,

c'est de montrer que l'algorithme se termine et fait bien ce que l'on attend de lui.

« Preuve de programme »

Phase 4: mesurer l'efficacité de l'algorithme indépendamment de l'environnement (machine, système, compilateur, …)

« Calcul de la complexité théorique »

Les phases à suivre pour analyser un algorithme

La complexité théorique de l'algorithme est calculée 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

Les paramètres de l'analyse de l'algorithme

- *La taille des entrées*
- *La distribution des données*
- *Les structures de données*

Les phases à suivre pour analyser un algorithme

Phase 5: mise en œuvre → programmation

évaluation expérimentale de l'algorithme :

« Calcul de la complexité expérimentale »

Phase 6:

Question:

Existe-il un algorithme qui résout le même problème P?

Evaluation d'un algorithme

Etant donnés deux algorithmes qui calculent les solutions à un même problème. Comment comparer ces algorithmes? Autrement dit, quel est le meilleur?

Intuition:

On préférera celui qui nécessite le moins de ressources

- Ressource de calculs
- Ressource d'espace de stockage;

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

Objectifs du calcul de la complexité

- Pouvoir prévoir le temps d'exécution d'un algorithme;
- Pouvoir comparer deux algorithmes réalisant le même traitement.

Calcul de la complexité d'un algorithme

Complexité temporelle

- Approche théorique
- Approche expérimentale

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 expérimentale

- 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 Asymptotique

Calcul de la complexité théorique

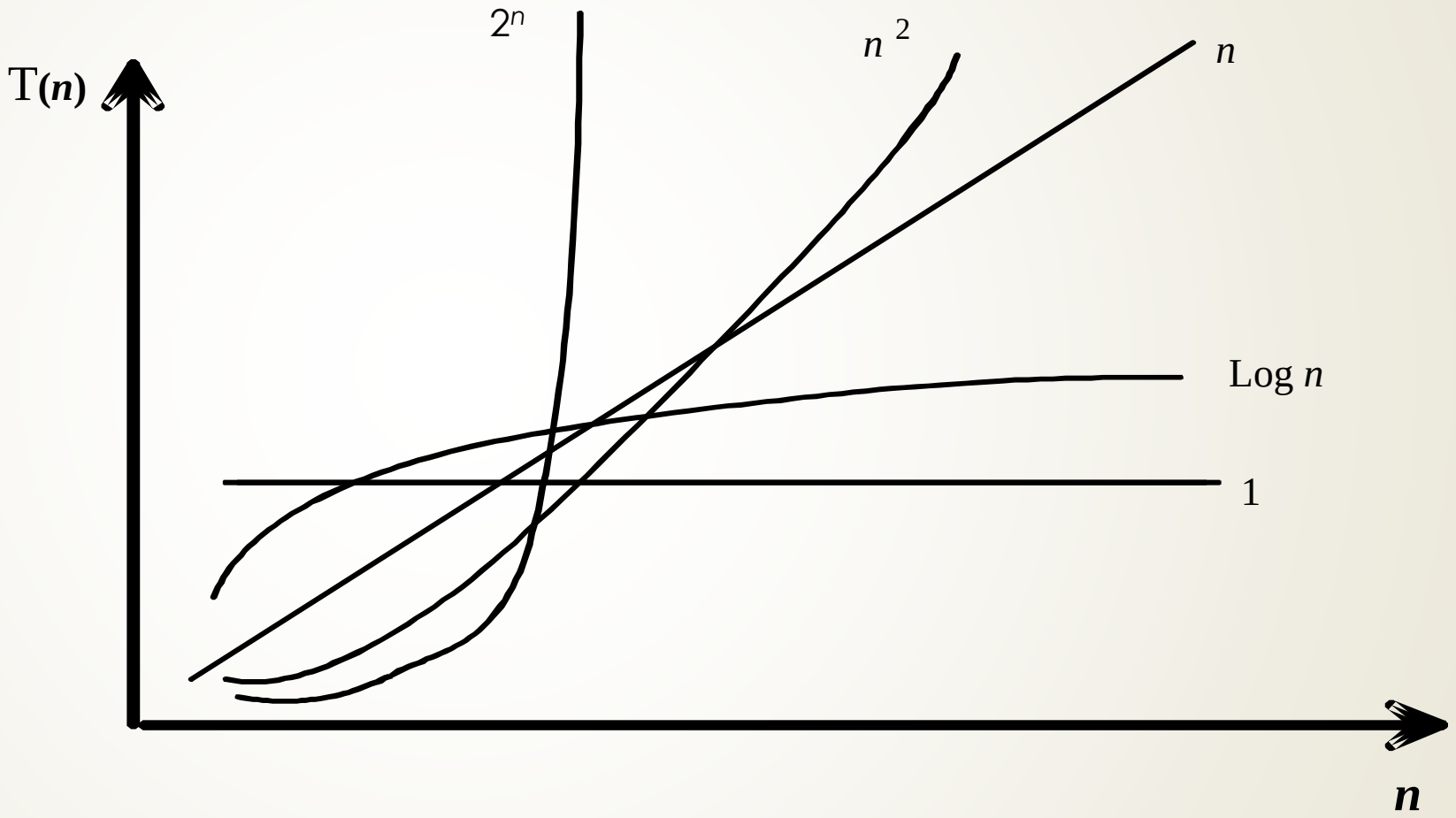
Les données des algorithmes sont en général de grande taille et qu'on se préoccupe surtout de la croissance de cette complexité en fonction de la taille des données.

On va donc considérer le comportement à l'infini de la complexité.

Le comportement d'un algorithme quand la taille des données devient grande s'appelle :

Complexité Asymptotique

Ordres de grandeur *1, log n, n, n^2, 2^n*



Complexité théorique

Asymptotique:

- **n** (taille des données) de grande taille
- Complexité approximative
- On retient le terme de poids fort de la formule
- On ignore le coefficient multiplicateur.

Les types de complexité

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

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 $n\,! >> 2^n$

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 thêta)

La notation ~ (de l'ordre de ; équivalent à)

Les notations Landau

- **Notation grand O :**

$$O(g(n)) = \{ f : IN \rightarrow IN \mid \exists\, c>0 \text{ et } n_0 \geq 0$$

$$\text{tels que } \mathbf{0 \leq f(n) \leq c.g(n)} \quad \forall n \geq n_0 \ \}$$

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

$$\mathbf{f(n) = O(g(n))}.$$

Notation O

- *f(n) est de l'ordre de* $O(g(n))$

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

Exemple:

$$f(n) = 2n + 5 \text{ est de l'ordre de } \boldsymbol{O(n)}$$

$$car\ 2n + 5 \leq 3n \qquad \forall n \geq 5$$

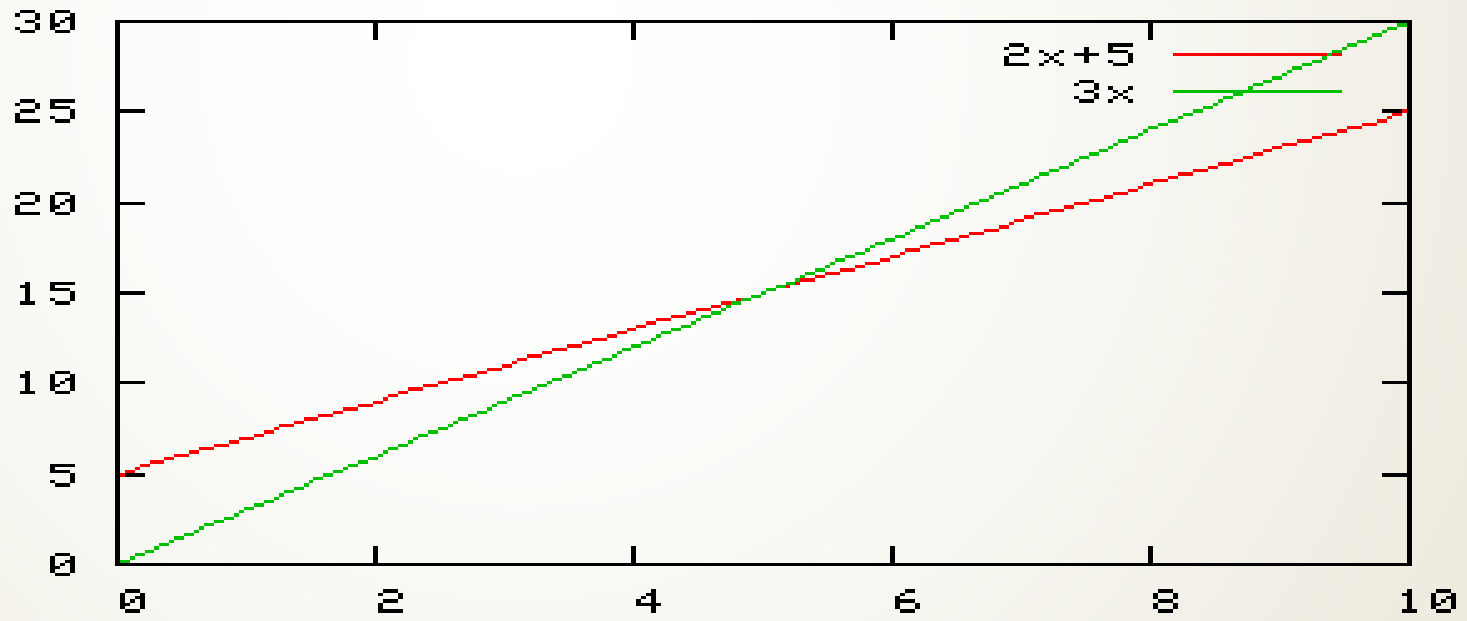
$$d'où\ c = 3\ et\ n_0 = 5$$

Notation O

Exemple : $f(n) = 2n+5$ est de l'ordre de $O(n)$

car $2n+5 \leq 3n \quad \forall n \geq 5$

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 $\mathbf{0 \leq c.g(n) \leq f(n)}$ $\forall n \geq n_0$ $\}$

- On *dit que g(n) est une* ***borne inférieure asymptotique*** *pour f(n), on note abusivement*

$$f(n) = \Omega(g(n)).$$

Notation Ω

- *f(n) est de l'ordre de* $\Omega(g(n))$

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

Exemple:

$$f(n) = 2n + 5 \text{ est de l'ordre de } \mathbf{\Omega(n)}$$

$$car\ 2n + 5 \geq 2n \qquad \forall n \geq 0$$

$$d'où\ c = 2\ et\ n_0 = 0$$

Les notations Landau

Notation Θ

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

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

$$f(n) = \Theta(g(n))$$

Notation Θ

- *f(n) est de l'ordre de* $\Theta(g(n))$

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

$\mathbf{0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)}$ $\forall n \geq n_0 \}$ *g(n) borne asymptotique*

Exemple:

$f(n) = 2n + 5$ *est de l'ordre de* $\boldsymbol{\Theta(n)}$

$car \quad 2n \leq 2n + 5 \leq 3n \qquad \forall n \geq 5$

$d'où \quad c_1 = 2, C_2 = 3 \; et \; n_0 = 5$

Les notations Landau

Remarque :

Notations o et ω avec des inégalités strictes.

Notations Landau

Exercice: Montrer que:

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

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

$$f(n)=n^2 = O(10^{-5}\ n^3)$$

$$f(n) = 10n^3 + 3n^2 + 5n + 1 = O(n^3)$$

Vocabulaire

- *O(1)* : temps constant, indépendant de la taille des données. C'est le cas le plus optimal.(Résolution de l'équation du second degré)
- O(n) : linéaire (La recherche d'une valeur dans un tableau)
- $O(n^2)$: quadratique (tri des éléments d'un tableau)
- $O(n^3)$: cubique (produit de deux matrices)
- $O(n^k)$: complexité polynômiale,
- O(log n), O(n log n),… complexité logarithmique
- $O(2^n)$, O(n !), … : complexité exponentielle, factorielle, …