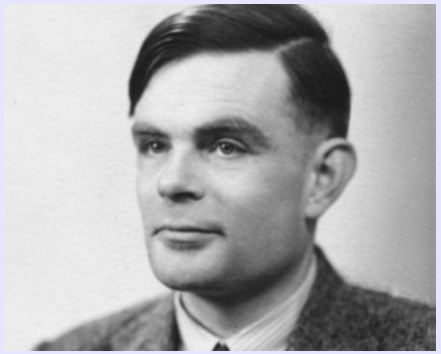




Information, Calcul et Communication

Module 1 : Calcul

Leçon 1.5 : Théorie du Calcul

Bernard Moret

les modifications de R. Boulic sont basées sur: "Computer Science, an overview" de G. Brookshear, Pearson, Cours de J-P Delahaye, Univ. de Lille, Cours de M. Betrema, Univ. de Bordeaux

Objectif du cours d'aujourd'hui

Etudier

les deux grandes questions de la théorie du calcul:

- *que peut-on résoudre avec un algorithme?*
- *que peut-on résoudre* efficacement *avec un algorithme?*

et les connexions avec

- *les mathématiques,*
- *l'information,*
- *la communication.*

Plan du cours d'aujourd'hui

- *Données + Question = Problème*
- *L'infini et l'énumération.*
- *L'incalculable sur la diagonale.*
- *Problèmes de décision*
- *Autoréférence et Problème de l'arrêt*

--

- *Prendre la mesure des problèmes.*
- *Calculer la réponse.*
- *Vérifier la solution.*
- *Conclusions*

Des données et une question

- Les leçons 2-3-4 ont présenté des algorithmes pour la recherche, le tri, et les chemins les plus courts.
- Ces algorithmes marchent *quelles que soient les données*.
- Chaque algorithme résout chaque fois *la même question*.
- Faire tourner l'algorithme sur une entrée nous donne
  - *la réponse à une instance du problème*
- Faire tourner un algorithme n'est pas la solution du problème:
  - *la solution, c'est l'algorithme lui-même*

Problèmes et problèmes intéressants

Definitions:

Un problème se compose d'une question et d'un ensemble d'instances.
Une solution pour un problème est un algorithme qui répond correctement à la question pour chaque instance possible.

Observation:

Si l'ensemble d'instances est fini, il est toujours possible de résoudre le problème en bâtissant une table de correspondances, la plus simple des structures de données, qui stocke la réponse pour chaque instance.
L'algorithme de solution consulte cette table et imprime la solution trouvée.
il n'y a pas de calcul!

Conclusion:

Seuls les problèmes avec des ensembles infinis d'instances demandent du travail: ce sont les seuls qui nous intéressent dans le cours d'aujourd'hui.

Apprendre à compter

Nous savons bien sûr tous compter: 1,2,3,4,5,...
Mais comment compter un ensemble infini ?

Outil: Les entiers positifs, N = {1,2,3,...}définissent le comptage.

Definition: un ensemble *S* est dénombrable
si et seulement si tout élément de *S* peut être numéroté par un entier,
c'est à dire qu'il existe une surjection $f : \mathbb{N} \to S$.

Exemple: *Si S est l'ensemble* $\mathbb{Z} = \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$.
C'est un ensemble dénombrable car on peut prendre
$f : \mathbb{N} \to \mathbb{Z}$ *comme suit:*

si l'entier naturel ***i*** *est pair,* alors $f(i) = i/2$
si l'entier naturel ***i*** *est impair,* alors $f(i) = -(i-1)/2$

Le numérotage est le suivant (entier naturel en bleu, $f(i)$ *en rouge):*
1, 2, 3, 4, 5, 6, 7, . . .
0, 1, -1, 2, -2, 3, -3, . . .

Les paires d'entiers positifs sont dénombrables

Sûrement pas! Après tout, il y a un nombre infini de paires qui ont toutes le même premier élément!

Et pourtant… Ecrivons les paires sur un tableau comme ci-dessous:

1, 1	1, 2	1, 3	1, 4	1, 5	. . .
2, 1	2, 2	2, 3	2, 4	2, 5	. . .
3, 1	3, 2	3, 3	3, 4	3, 5	. . .
4, 1	4, 2	4, 3	4, 4	4, 5	. . .
5, 1	5, 2	5, 3	5, 4	5, 5	. . .

Nous pouvons énumérer les paires sur la base de leur somme *car toute valeur de somme est obtenue à partir d'un nombre fini de paires.*

Donc nous commençons par (1,1), puis (1,2) et (2,1), puis (1,3), (2,2) et (3,1), puis (1,4), (2,3), (3,2) et (4,1), etc.

Chaque paire reçoit son numéro unique—cet ensemble est dénombrable.

Les programmes sont dénombrables

Un programme est simplement un texte (de longueur finie) écrit à l'aide d'un alphabet choisi. Il est facile d'énumérer (numéroter) tous les textes possibles dans un ordre lexicographique:

- Commençons par énumérer les textes d'un seul caractère:
  1: a, 2: b, 3: c, . . . , 26: z
- Continuons avec les textes de deux caractères:
  27: aa, 28: ab, 29: ac, . . . , 52: az, 53: ba, 54: bb, . . . ,
  677: za, 678: zb, 679: zc, . . . , 702: zz
- Maintenant passons aux textes de trois caractères:
  703: aaa, 704: aab, 705: aac, . . . , 18'278: zzz, et ainsi de suite.

Même avec le bon choix d'alphabet, certain des textes énumérés ne sont pas des programmes, mais tous les programmes sont énumérés.

Les fonctions N→N ne sont pas dénombrables

Soit l'ensemble des fonctions entières d'une variable entière n,
c'est à dire qui produisent une valeur entière ***f(n)*** pour tout entier ***n***.

Pour essayer de dénombrer ces fonctions, on va les ranger sous forme d'un tableau (infini) avec une fonction par ligne.

Chaque colonne ***j*** indique la valeur de la fonction pour l'entier ***j***

Exemple: la fonction f_i apparait sur la ligne ***i***

$$f_i(0),\ f_i(1),\ f_i(2),\ \ldots,\ f_i(j),\ \ldots$$

Les fonctions N→N ne sont pas dénombrables

	0	1	2		j	
0	$f_0(0)$,	$f_0(1)$,	$f_0(2)$,	... ,	$f_0(j)$,	...
1	$f_1(0)$,	$f_1(1)$,	$f_1(2)$,	... ,	$f_1(j)$,	...
2	$f_2(0)$,	$f_2(1)$,	$f_2(2)$,	... ,	$f_2(j)$,	...
3	$f_3(0)$,	$f_3(1)$,	$f_3(2)$,	... ,	$f_3(j)$,	...
	...					

On définit maintenant la fonction entière $\gamma(n) = f_n(n) + 1$

$\gamma(n)$ est construite en prenant les termes de la diagonale du tableau et en ajoutant 1

Conséquence: aucune ligne du tableau ne peut représenter $\gamma(n)$
car son terme diagonal ne serait pas correct
-> les fonctions ne sont pas dénombrables

Pas assez de programmes!

Qu'avons-nous appris?

L'ensemble de tous les programmes (quel que soit le langage) est dénombrable.

L'ensemble des fonctions entières d'une variable entière n'est pas dénombrable.

Donc il existe *beaucoup plus* de fonctions entières que de programmes.

Donc il existe un nombre infini de fonctions qui ne sont pas calculables par un programme!

(Il n'y a simplement pas assez de programmes possibles.)

Pas assez de programmes! (2)

Tout programme possède une longueur finie

Une fonction entière produit une valeur pour un nombre infini d'entiers

$f_i(0),\ f_i(1),\ f_i(2),\ \ldots,\ f_i(j),\ \ldots\ \ldots$

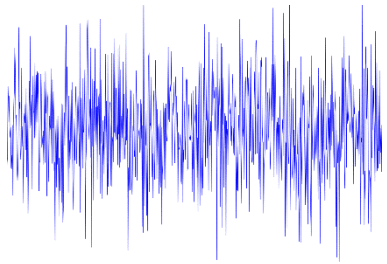
Exprimer la fonction sous forme d'une table demanderait une longueur infinie de table, ce qui n'est pas permis pour un programme

0 1 2 ... j ... ...
$f_i(0),\ f_i(1),\ f_i(2),\ \ldots,\ f_i(j),\ \ldots\ \ldots$

On ne peut pas exprimer tous les problèmes sous forme algébrique.

$f_i(n) = a.n + b$

Par exemple, les fonctions *aléatoires* dont les valeurs successives sont imprévisibles ne peuvent pas être produites par un programme dont la longueur est finie



Problème de décision et décidabilité

Définition: un problème de *décision* prend la forme d'une question
à laquelle on peut seulement répondre par OUI ou par NON

Le problème est décidable si l'on peut apporter l'une ou l'autre réponse à l'aide d'un algorithme opérant en un nombre fini d'étapes.

Le problème est indécidable s'il est impossible de répondre à l'aide d'un algorithme.

exemple de problème décidable

Soient n et m deux entiers donnés >1. m est il multiple de n?

On sait que: "8 est multiple de 2" est VRAI
"16 est multiple de 3" est FAUX

Deux instances du problèmes ne suffisent pas pour déterminer si le problème est décidable ou pas. Il faut le montrer pour tout m et n.

L'algorithme suivant permet de répondre pour tout m et n:

- **faire la division entière de m par n pour obtenir le reste r**
- **Si le reste r est nul répondre VRAI**
- **Sinon répondre FAUX**

ce problème est décidable

exemple de problème indécidable: le problème de l'arrêt

Definition: **déterminer si un programme donné P(x), initialisé avec une valeur de x (appartenant à N), parvient à s'arrêter ou non.**

Exemple trivial: soit le programme P(x)

```
Tant que x > 0
    incrémenter x
```

Si le programme est exécuté avec une valeur initiale nulle pour x, celui-ci se termine immédiatement

Si par contre il est initialisé avec x strictement positif, il boucle sans fin.

Outil de démonstration: l'autoréférence

Il y a **autoréférence** lorsqu'un signe se réfère à lui-même.

La phrase ***"cette phrase compte cinq mots"*** est autoréférente.

Les phrases autoréférentes peuvent être paradoxales:

"cette phrase est un mensonge"

Elle ne peut etre classée ***ni vraie***, ***ni fausse***
(~ paradoxe d'Epiménide, ~2500 BP)

Outil de démonstration: l'autoréférence (2)

L'autoréférence est utilisée dans la démonstration de l'indécidabilité du problème de l'arrêt d'un programme P(x) en posant qu'on s'intéresse seulement au contexte suivant :

la valeur de x est un entier qui représente le programme P(x) lui-même.

Exemple trivial: soit le programme P(x)

```
Tant que x > 0
    incrémenter x
```

Par construction (vue en début de cours) la valeur entière de x représentant le programme P(x) est strictement supérieure à 0, ce qui conduit le programme à boucler sans fin.

Indécidabilité du problème de l'arrêt (1)

Nous allons montrer qu'il ne peut pas exister d'algorithme général **Q(x)** qui, implémenté sous forme d'un programme Q(x), puisse déterminer si n'importe quel programme P(x) s'arrête ou pas lorsqu'il est exécuté avec lui-même comme valeur initiale x.

Démonstration par l'absurde:

Etape 1

Supposons que Q(x) existe. Cela veut dire que :

- Q(x) est exécuté avec x représentant le programme P(x)
- Q(x) détermine si P(x), initialisé avec lui-même comme valeur de x, s'arrête ou pas
- Q(x) renvoie seulement deux valeurs possibles:

    1 pour "OUI P(x) peut s'arrêter quand il est initialisé avec lui-même"

    0 pour "NON P(x) ne peut PAS s'arrêter quand il est initialisé avec lui-même"

Indécidabilité du problème de l'arrêt (2)

Etape 2

Construisons un nouveau programme R(x) qui contient Q(x) et qui utilise la valeur renvoyée par Q(x), notée y:

```
y ← Q(x)
Tant que y > 0
    incrémenter y
```

R(x)

Etape 3

Que se passe-t-il quand ce nouveau programme R(x) s'exécute lorsqu'il est initialisé avec lui-même ?

- supposons que R(x) se termine, donc **Q(x) renvoie la valeur 1** ce qui produit une boucle sans fin sur la variable y, donc **R(x) ne peut PAS se terminer** [contradiction]

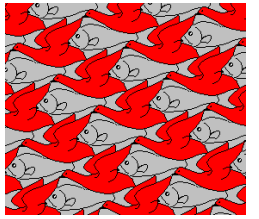
- supposons que R(x) ne se termine pas, donc **Q(x) renvoie la valeur 0** ce qui fait immédiatement quitter la boucle sur y, **donc R(x) se termine** [contradiction].

Conclusion: l'existence de Q est impossible

Autres problèmes indécidables

1) un programme contient-il un morceau de code inutile? (Th. de Rice)
2) deux programmes calculent-ils la même chose ? (Th. de Rice)
3) une configuration du jeu de la vie de Conway finit-elle par disparaitre ou persiste-t-elle toujours ?
4) peut-on paver le plan sans recouvrement ni espace vide avec un ensemble de formes géométriques ?

http://www.michael-hogg.co.uk/game_of_life.php

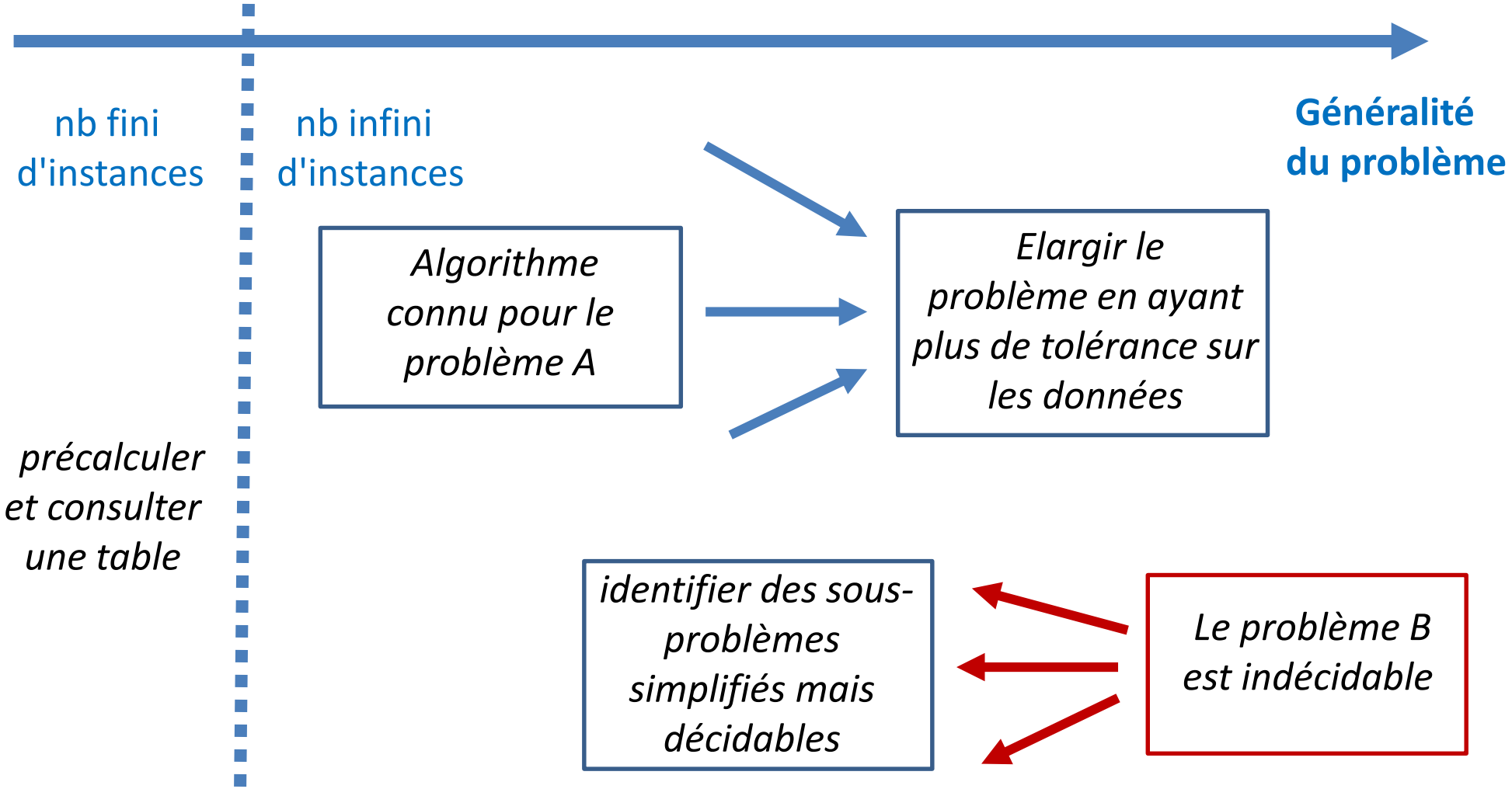


Maurits Cornelis Escher utilisé dans:
http://xavier.hubaut.info/coursmath/doc/pavages.htm

Quand un problème est indécidable, on SAIT qu'on ne peut pas déterminer de réponse à la question posée <=> il est inutile de perdre du temps sur cette formulation d'un problème ou de croire quelqu'un qui voudrait nous vendre une solution à ce problème...

Cela est beaucoup plus fort que de dire qu'on ne sait pas le résoudre (simple aveu d'ignorance <=> quelqu'un d'autre pourrait réussir)

Que faire en cas de problème indécidable?



En résumé

La plus grande partie des problèmes que l'on peut définir par les mathématiques n'ont pas de solution algorithmique.

L'aspect clef de ces problèmes est l'autoréférence, un outil (presque trop) puissant en informatique.

La plupart de ces problèmes sont artificiels et n'ont aucun intérêt, mais certains sont importants dans la pratique, tels que des questions de caractéristiques des programmes.

Cependant, tout n'est pas perdu: il peut exister des algorithmes qui résolvent les instances courantes de ces problèmes.

Plan du cours d'aujourd'hui

- *Données + Question = Problème*
- *L'infini et l'énumération.*
- *L'incalculable sur la diagonale.*
- *Problèmes de décision*
- *Autoréférence et Problème de l'arrêt*

--

- *Prendre la mesure des problèmes.*
- *Calculer la réponse.*
- *Vérifier la solution.*
- *Conclusions*

La complexité: que mesurer et comment?

Certaines tâches d'informatique doivent se faire des millions de fois par seconde dans le monde entier (transactions bancaires); d'autres demandent des milliards de milliards d'opérations (modèles climatiques); enfin d'autres se font sur des données gigantesques (Google, génome humain).

La complexité d'un problème est celle du meilleur algorithme qui le résout.

La complexité n'est pas la *difficulté* du problème qui est un critère seulement valable pour un être humain. La complexité est une fonction de la *taille du problème* qui mesure son influence sur le temps de calcul et l'espace mémoire.

Exemple: la complexité du problème du tri est en O(nlog(n)), et il est démontré qu'on ne peut pas faire mieux

La complexité: une hiérarchie

La théorie du calcul partage les problèmes entre ceux qui peuvent et ceux qui ne peuvent pas être résolus par le calcul.

La théorie de la complexité n'étudie que les problèmes pour lesquels un algorithme existe et partage ces problèmes en fonction des demandes de temps ou d'espace de l'algorithme le plus efficace possible pour chaque problème.

Ce partage se fait en grandes classes de complexité, chacune un ensemble infini de problèmes pour lesquels l'algorithme le plus efficace fait à peu près les mêmes demandes de ressources (temps ou espace).

Le résultat est une hiérarchie de classes de complexité.

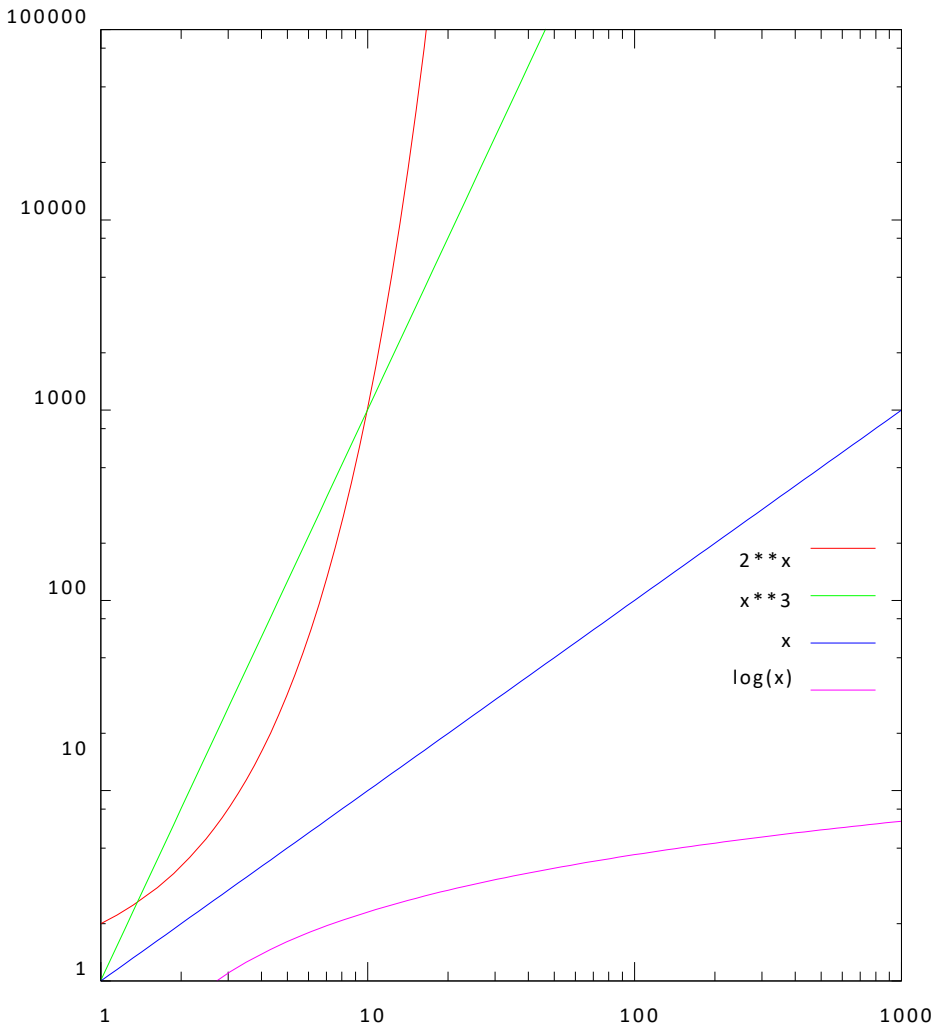
Quelles classes devrions-nous définir?

Algorithmes “pratiques"

Nous avons étudié dans les leçons précédentes des algorithmes de recherche, de tri, et pour trouver le chemin le plus court. Leurs temps de calcul variaient de temps linéaire à temps quadratique; de même pour leurs demandes d’espace.

En général, un algorithme “pratique" prend au plus un temps de calcul cubique (en fonction de la taille de l’entrée N) que l’on va généraliser à temps polynomial.

Parenthèse: Par contre, pour le volume de données de Google, CERN, ou NASA, un temps cubique est exclu: un temps linéaire est requis.



Algorithmes “pratiques"

Exemple:

un ordinateur moderne fait de l’ordre de 10 milliards (10^{10}) d’opérations par seconde.

Le CERN génère près d’un petabyte (N=10^{15} bytes) de données par semaine.

Un algorithme quadratique en $O(N^2)$ ferait au moins 10^{30} opérations sur ces données, demandant de l’ordre de 10^{30} / 10^{10} = 10^{20} secondes—ou 3 milliards de siècles!

Même un algorithme linéaire ferait au moins 10^{15} opérations, demandant 10^5 secondes, ou à peu près 30 heures.

Résoudre en temps polynomial: P

La classe de complexité devenue synonyme de solution efficace est la classe *P*, c'est-à-dire la classe des problèmes pour lesquels il existe un algorithme qui résout n'importe quelle instance de ce problème en temps polynomial.

Vu que les ordinateurs modernes ne peuvent pas utiliser plus qu'un nombre constant de bytes de stockage à chaque instruction, un temps polynomial implique un espace polynomial.

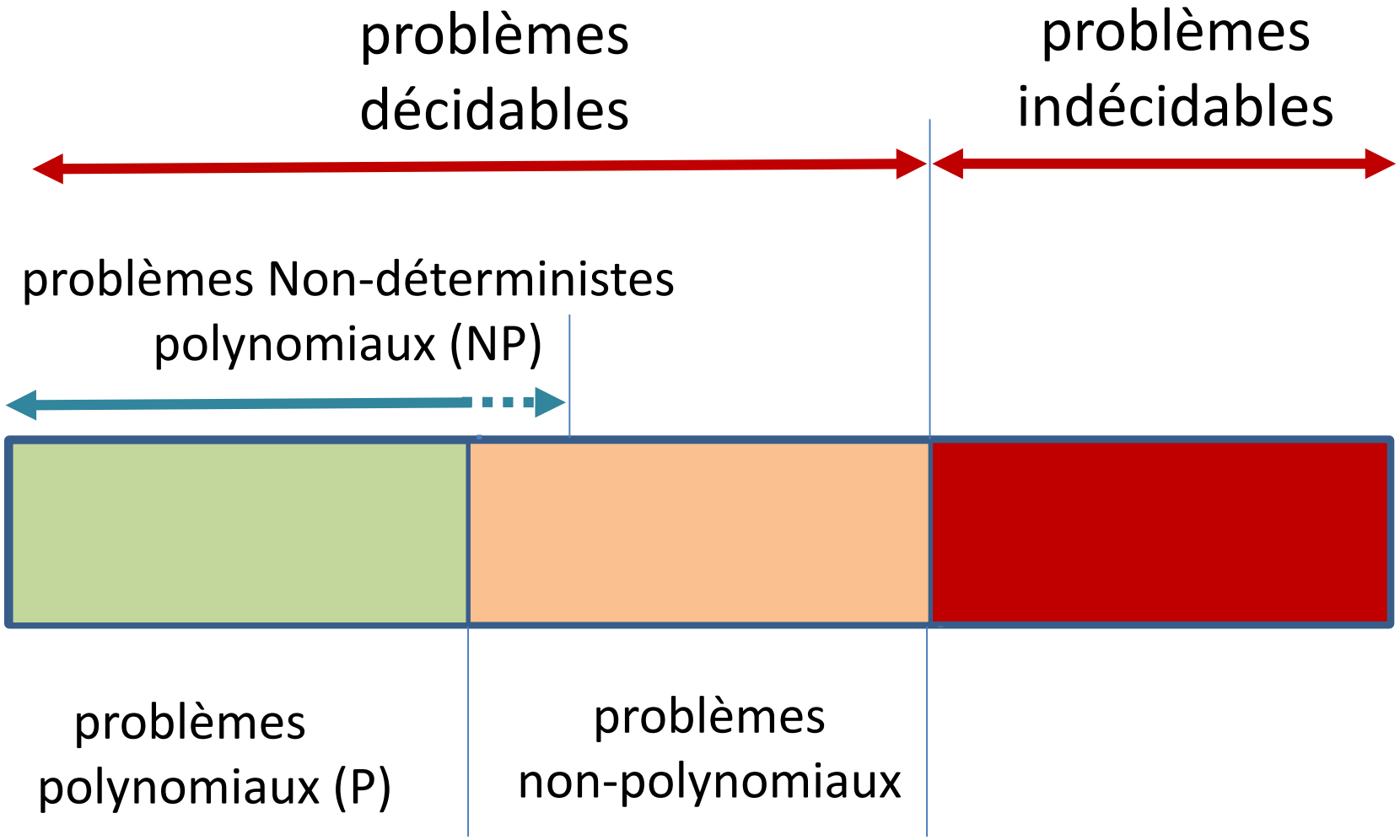
Bien entendu, ces polynômes sont typiquement cubiques ou plus petits: personne n'a conçu d'algorithme sérieux demandant un temps en n^{17} *…*

Exemples de problèmes dans P

Ouvrez n'importe quel livre sur les algorithmes: vous trouverez des centaines de pages décrivant des problèmes dans la classe *P* et leurs algorithmes de solution.

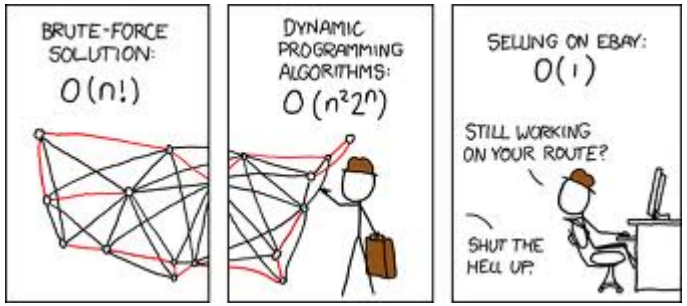
- la recherche par dichotomie: $\log n$
- trouver tous les amis dans un réseau social: n
- l'intersection de deux polygones convexes: n
- le tri: $n \log n$
- le chemin le plus court dans un graphe: $n \log n$
- l'alignement de deux séquences d'ADN: n^2
- l'affectation du personnel de bord aux vols de ligne: $n^2 \log n$
- la multiplication de deux matrices carrées the taille $n \times n$: $n^{2.37}$
- l'optimization du débit entre deux points d'un réseau: n^3

petite classification des problèmes [G. Brookshear]



Premier exemple de problème non-polynomial

Considérons le problème du voyageur de commerce qui doit visiter ses clients dans N villes sans dépasser son budget de déplacement. Il doit trouver un chemin qui part de chez lui, relie les N villes et revient chez lui, sans dépasser une distance maximum.



xkcd.com

Si on résout ce problème d'une manière systématique, on va d'abord considérer les N villes comme première destination, puis les (N-1) villes restantes, etc.. Le coût calcul est en O(N!) ce qui revient à un **coût exponentiel** avec la formule de Stirling. **Ce problème est donc non-polynomial.**

Cependant, si on "*dispose*" d'un trajet reliant les N villes (hasard, critère local ou autres heuristiques), il est **FACILE de VERIFIER** que ce trajet ne dépasse pas la distance maximum imposée. **Le problème de vérification devient linéaire, il appartient à P.**

Le problème est **Non-déterministe Polynomial (NP)** car l'exécution de cette méthode peut conduire à des solutions différentes pour les mêmes données du problème.

Exemple de construction d'une solution non-déterministe

Voyageur de Commerce (suite): l'algorithme suivant construit une solution non-déterministe (qui exploite le hasard) et la vérifie en temps polynomial:

On suppose qu'on dispose d'un algorithme **Random(K)** de génération d'un nombre *entier* **pseudo-aléatoire** compris entre **1** et **K**, chacun de ces nombres ayant la même probabilité de génération. On suppose que cet algorithme a un coût constant.

Soit **L** la liste des **N** villes .
Soit **P** une liste de taille **N** destinée à recevoir le plan de Parcours des N villes, c'est-à-dire que P(1) contiendra la première ville, P(2) la seconde etc…

Chaque ville **L(j)** utilisée pour **P** est remplacée par une valeur «**used**».

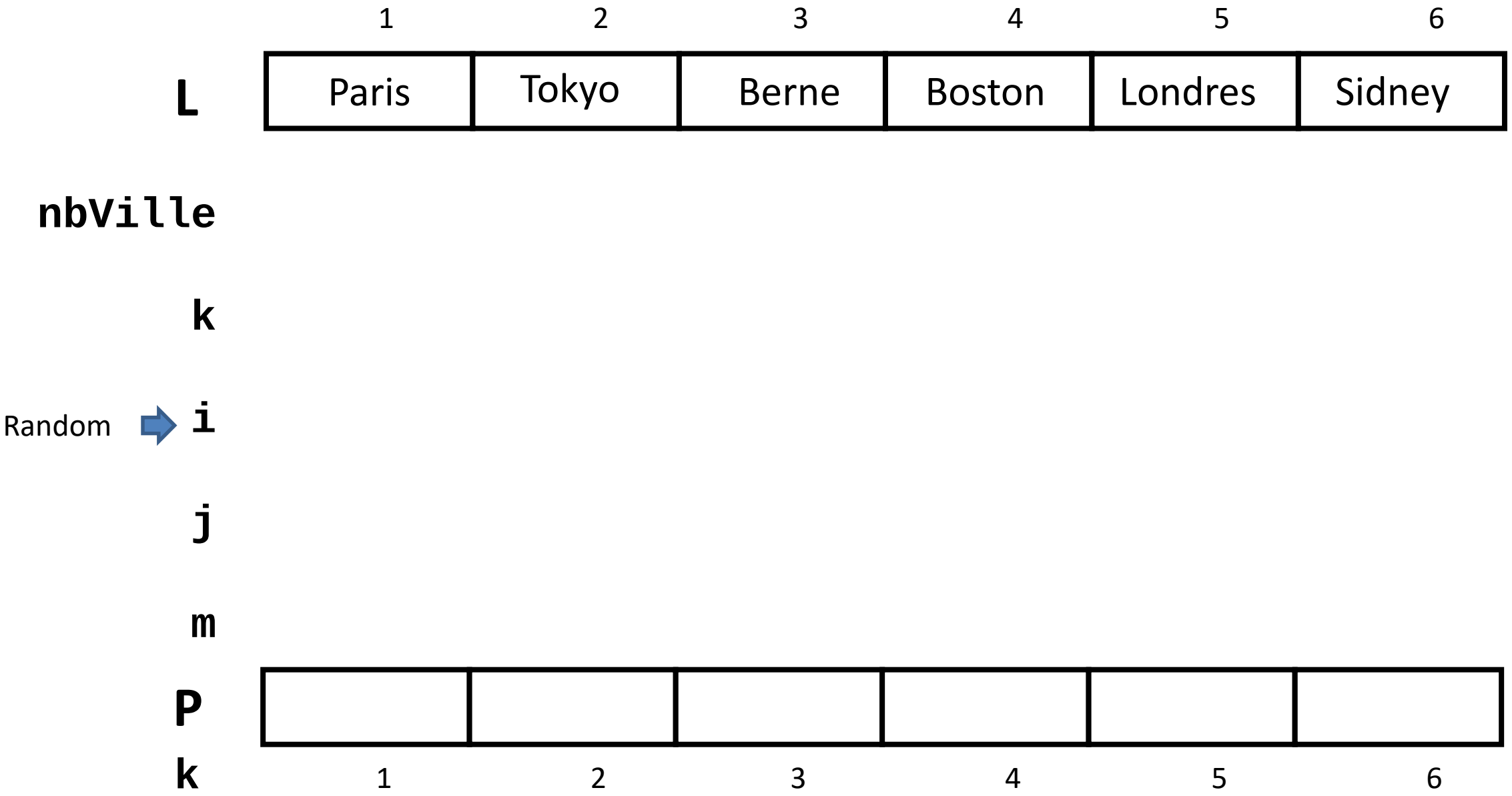
Enfin, on suppose connues les distances **D** entre chaque paire de villes.

Création et vérification solution Non-det

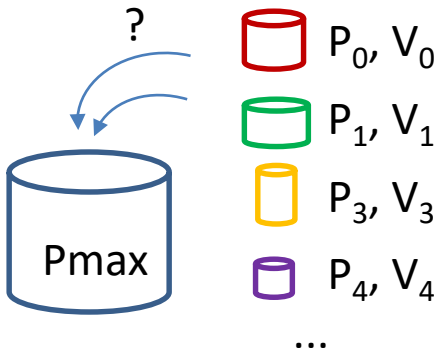
entrées : Liste non vide **L**, taille N, distance **dmax** > 0
sortie : Liste **P** de taille N, booléen distance(**P**) < dmax

```
nbVille ← N
Pour k de 1 à N                          // création parcours
    i ← Random(nbVille)                  // numéro de ville libre
    j ← 1
    m ← 1
    Tant que L(j) = «used» ou m < i      // filtre les villes «used»
        j ← j + 1                        // jusqu'à atteindre
        Si L(j) ≠ «used»                 // la iième ville libre
            m ← m+1
    P(k) ← L(j)
    L(j) ← «used»
    nbVille ← nbVille -1

distance ← 0
Pour k de 1 à N -1                       // calcul distance
    distance ← distance + D(P(k), P(k+1))
distance ← distance + D(P(N), P(1))
Sortir : distance < dmax
```


1
2
3
4
5
6
L
Paris
Tokyo
Berne
Boston
Londres
Sidney
nbVille
k
Random
i
j
m
P
k
1
2
3
4
5
6

second exemple de problème non-polynomial et NP (1)



Considérons le problème du sac à dos qui possède une capacité limitée à un poids Pmax. On désire y ranger des objets $\{O_i\}$, chacun caractérisé par son poids P_i et sa valeur V_i, telle que la ***valeur*** *totale rangée dans le sac est la plus grande possible*, sans que le poids total dépasse Pmax.

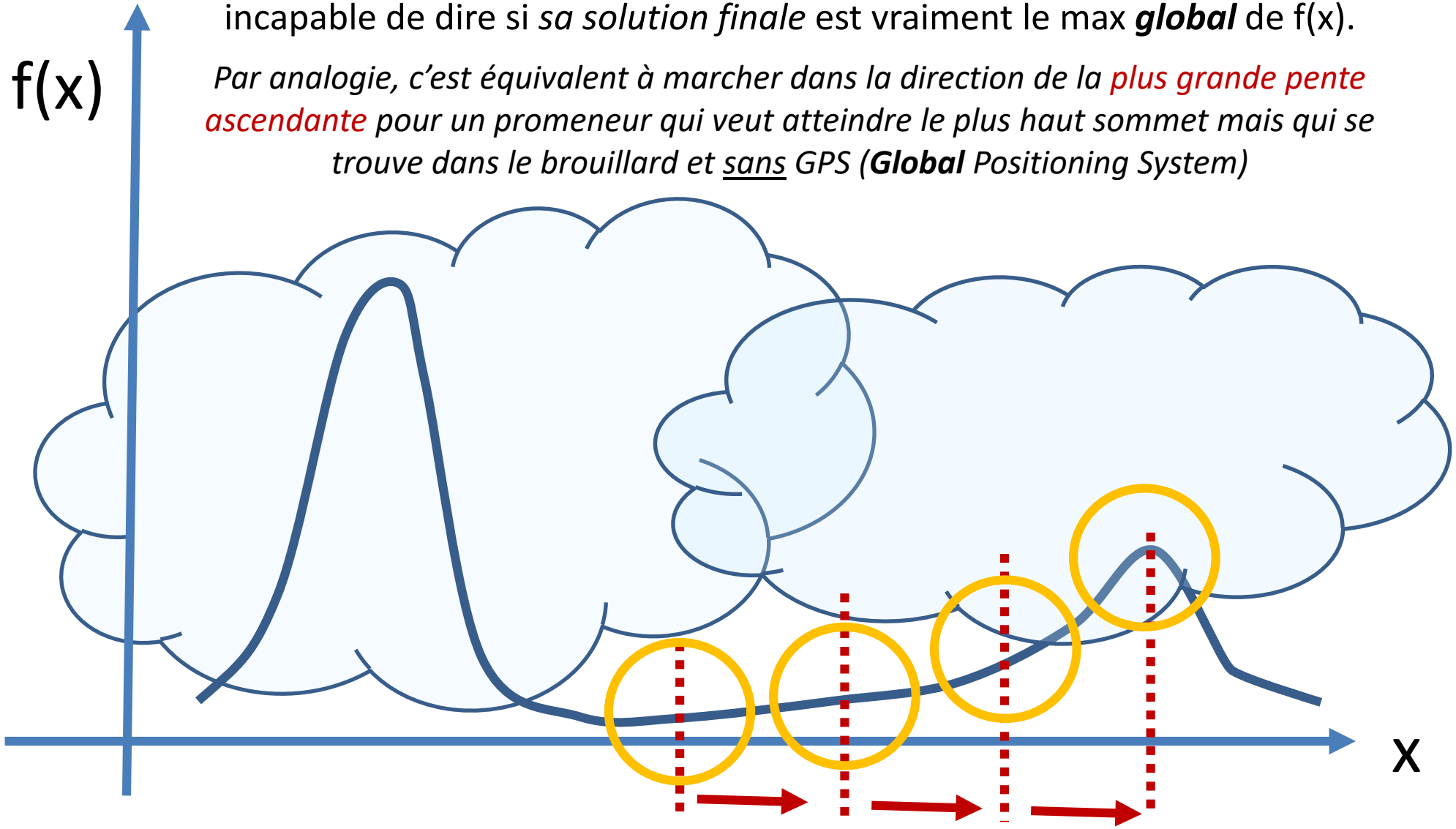
Si on résout ce problème d'une manière systématique, on va examiner tous les groupes possibles d'objets. Pour chaque objet il existe deux possibilités: soit il fait partie du groupe, soit il n'en fait pas partie. Donc s'il y a N objets, on peut construire $\mathbf{2^N}$ groupes différents: c'est un **complexité exponentielle**. Ce problème est **non-polynomial**.

Cependant, le problème appartient à NP car on peut construire ET **VERIFIER** une solution en utilisant un algorithme à **coût polynomial** de type ***glouton.*** Ce type d'algorithme fait le choix d'inclure un objet dans le sac à dos sur la base d'une décision qui ne sera jamais remise en question ultérieurement. Cela conduit généralement à un optimum local comme illustré dans le prochain slide.

dans quelle direction l'algo glouton effectue-t-il sa recherche ?

L'algorithme **glouton** exploite un critère ***local*** facile à calculer ; de ce fait il est incapable de dire si *sa solution finale* est vraiment le max ***global*** de f(x).

*Par analogie, c'est équivalent à marcher dans la direction de la plus grande pente ascendante pour un promeneur qui veut atteindre le plus haut sommet mais qui se trouve dans le brouillard et sans GPS (**Global** Positioning System)*



second exemple de problème non-polynomial et NP (2)

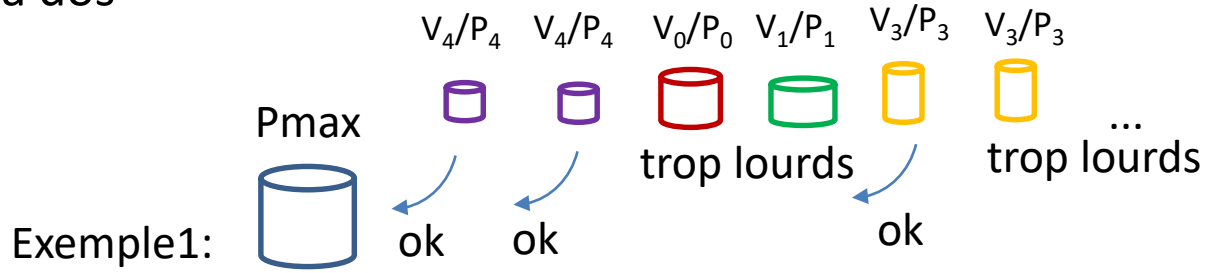
Problème du sac à dos (suite): l'algorithme glouton utilisé pour obtenir une solution (non-optimale) procède comme suit:

Trier les objets selon leur rapport **Valeur/Poids**

Pour chaque objet trié, en commençant par celui de rapport **V/P le plus grand**

Si le poids de l'objet produit un poids total < Pmax

ajouter l'objet au sac à dos



Malgré la bonne décision "locale" pour le choix d'inclure un objet, l'algorithme glouton peut conduire à une solution loin d'être optimale. Supposons un sac avec **Pmax=3** et trois objets de **(Poids , Valeur)** : **(2 , 5), (1.5 , 3), (1.5 , 3)**. Le premier objet possède le meilleur ratio **V/P** et est donc choisi mais ensuite il ne reste plus de place pour les 2 autres. La valeur totale est **5** alors qu'on aurait eu une valeur totale de **6** (3+3) en mettant les 2 autres objets.

Vérifier en temps polynomial: NP

La classe de complexité caractérisant les problèmes dont les solutions peuvent être vérifiées efficacement (en temps polynomial) est la classe *NP*.

Du point de vue pratique, ce sont les seuls problèmes qui vaillent la peine de résoudre: à quoi servirait une solution qui n'est même pas vérifiable?

> Remarque: *La différence entre P et NP (tout deux définis en termes de temps polynomial) est celle entre résoudre (calculer une solution) et vérifier (une solution donnée).*

NP contient P — car tout algorithme polynomial (déterministe) peut contenir une instruction non-déterministe qui n'affecte pas ses performances.

Mais résoudre est apparemment plus difficile que vérifier: nous connaissons des centaines de problèmes dans la classe NP pour lesquels nous n'avons pas pu trouver d'algorithme de solution.

Le plus grand problème théorique d'aujourd'hui (en informatique et en math):

est-ce que *P* est égal à *NP*?

Exemples de problèmes dans NP

- Re-ouvrez ce livre sur les algorithmes: vous trouverez des dizaines de pages de plus, décrivant des problèmes dans la classe *NP*—moins de pages que pour *P*, vu qu'il n'y aura pas d'algorithmes pour ces problèmes.

  problèmes d'optimisation sur les graphes: couvertures, coloriages, circuits, etc.

  problèmes sur séquences (texte ou ADN)

  satisfaire une formule logique, équivalence de formules
  problèmes de partitions en sous-ensembles (rendre la monnaie, affectations diverses, etc.)
  problèmes de regroupement
  problèmes d'ordonnancement
  problèmes de positionement
  problèmes en mathématiques, physique, biologie, etc.

- Presque tous les problèmes d'optimisation d'importance en industrie sont dans cette classe—et bien trop peu d'entre eux se trouvent dans *P*!

Conclusions

Notre capacité moderne d'accumuler des quantités énormes de données et notre ambition de comprendre des systèmes de plus en plus complexes (le génome, le climat, l'écologie, le cerveau) rendent nécessaire la maîtrise de la complexité et des techniques de vérification.

L'informatique théorique nous indique ce qui est possible et ce qui ne l'est pas et nous guide dans le développement de solutions.

Fin du module 1: Calcul

La semaine prochaine: début du module sur les fondements de la communication