Algorithmique et Complexité

- classification des problèmes/algorithmes
- degrés de complexité
- P = NP ?
- approche descendante
- diviser our résoudre

May 09, 2001   12:52

Algorithme

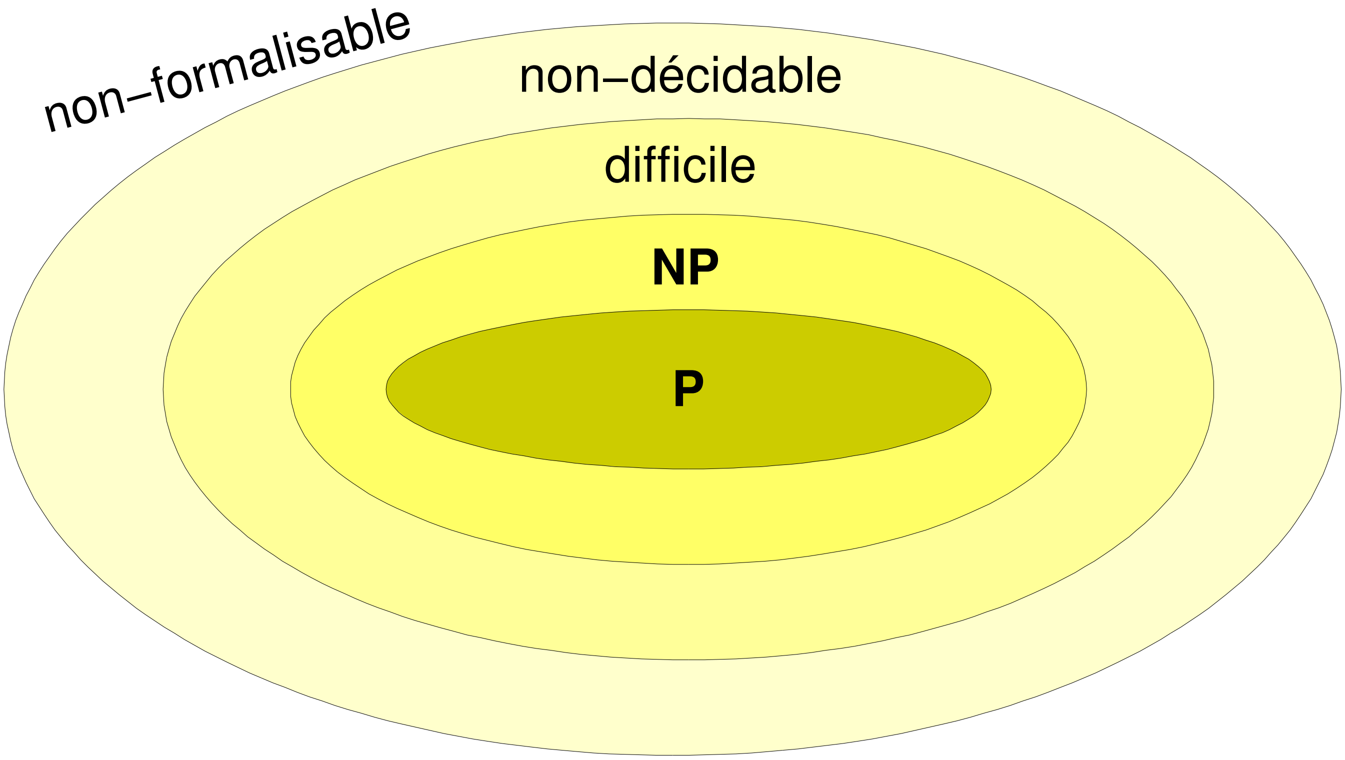
... est

une **suite finie** de règles à appliquer,
dans un **ordre déterminé**,
à un **nombre fini de données**,

pour arriver, en un **nombre fini d'étapes**, à un résultat,
et cela quelque soient les données traitées.

[Encyclopedia Universalis]

Le Monde des Problèmes



Les Problèmes Non Formalisables

Des problèmes:

- qui sont **intrinsèquement vagues**

  La vie a–t–elle un sens ?
  Dieu existe–t–il ?

- pour lesquels il est difficile d'obtenir un **consensus** sur leur **spécification précise**

  Déterminer la santé financière d'une entreprise.

Les Problèmes Non Décidables

On ne peut pas résoudre tous les problèmes formalisables.

Exemple:

Halting problem !

Problème bien et clairement défini.
Mais, il n' existe pas de solution !

Vers Les Problèmes NP–complets

La notion de **complexité** donne un moyen de *comparer* les solutions algorithmiques pour un problème décidable selon le *temps d'exécution* et la *quantité de mémoire* requise lors de la mise en oeuvre de ces algorithmes.

Ici, on va se concentrer sur le premier critère.

On mesure le temps d'une façon indépendante de tout implantation en calculant le *nombre d'étapes exécutées*.

Entrées et Temps d'Exécution (I)

La complexité algorithmique **c** d'un problème, c'est
la relation entre la taille/le nombre **n** des données
et le nombre d'étapes **O(c(n))** requises par un algorithme.

[Landau–O]

Par exemple:

O(n) complexité ***linéaire***

Signification:

Un algorithme a la complexité **O(n)**,
si on peut donner une fonction **f** de la forme **f(n) = k*n**
pour une constante **k** qui borne **le pire cas du nombre d'étapes requises** par l'algorithme pour «**n** *grand*».

Entrées et Temps d'Exécution (II)

O(log n) complexité ***logarithmique***
très bien (faible croissance ...)

O(n) complexité ***linéaire***
bien (durée fixe pour chaque argument)

O(n log n) complexité ***légèrement sur–linéaire***
pas mal (diviser pour résoudre)

O(n^2) complexité ***quadratique***
pas bien (p.e., boucles imbriquées)

O(n^3) complexité ***cubique***
*pas bien du tout (seulement pour **n** petit)*

O(2^n) complexité ***exponentielle***
catastrophique! (tous les possibilités)

O(n!) complexité ***factorielle***

Polynomial et Non–Polynomial

problèmes **faciles**:
  problèmes de complexité polynomiale
  admettant des algorithmes «efficaces»
  (solutions raisonnables)

problèmes **difficiles**:
  problèmes de complexité non–polynomiale
  n'admettant que des algorithmes «inefficaces»
  (solutions pas raisonnables)

Nombre d'Etapes «Turing–Pair»

Soit n le nombre de chiffres du nombre b,
c–à–d, si b=1001, alors n = 4.

$g(n) = n + 1 + n + 1 = \boxed{\mathbf{2*n} + 2}$

1) pour trouver la fin du nombre, en allant vers la doite
2) changement de direction
3/4) pour trouver le début du nombre, en allant vers la gauche
5) pour déposer le résultat

Donc, cet algorithme a la ***complexité linéaire*** **O(n)**.

Nombre d'Etapes à Hanoi

$f(1) = 1$

$f(2) = 2 * f(1) + 1 = 2*1 + 1$

$f(3) = 2 * f(2) + 1 = 2*(2*1 + 1) + 1$

$f(4) = 2 * f(3) + 1 = 2*(2*(2*1 + 1) + 1) + 1$

...

$f(n) = 2 * f(n-1) + 1 = 2^{n-1} + 2^{n-2} + \ldots 2^2 + 2^1 + 1 = \boxed{\mathbf{2^n - 1}}$

Alors, cet algorithme a la ***complexité exponentielle*** **$O(2^n)$**.

Donc, si on a N=64 anneaux
(comme les Tibétains l'ont posé comme problème),
on a besoin d'exécuter $1{,}8 * 10^{19}$ mouvements.
Même avec 1000000/sec, cela prendrait 584942 ans ...

Les Problèmes de Décision

... sont des problèmes qui peuvent s'exprimer sous la forme d'une question attendant une réponse binaire (*oui* ou *non*).

1) description de l'instance du problème
2) l'expression d'une question décisive sur cette instance

Exemple («Halting problem»):

1) l'ensemble de tous les algorithmes
2) existe–t–il un entre eux qui va décider de tout autres s'ils vont se terminer ou pas?

A côté des problèmes non décidables, il y a aussi des problèmes décidables, mais la décision peut être difficile ...

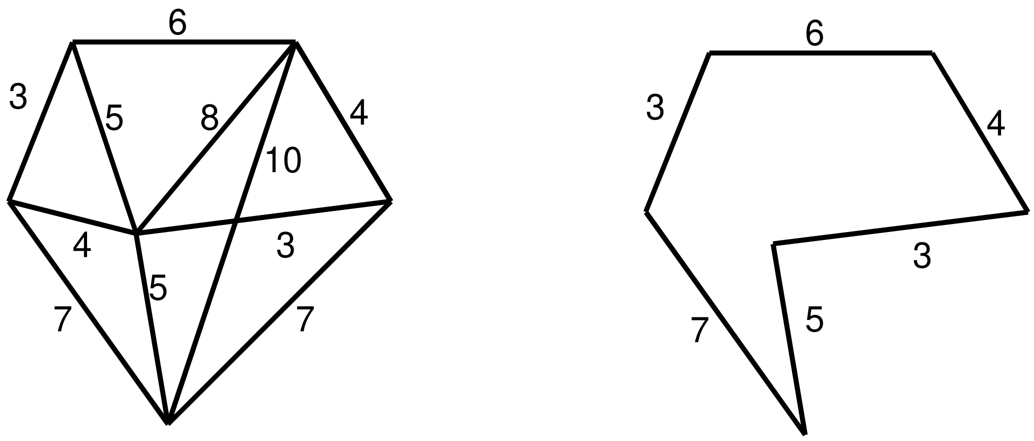
Exemple: «Voyageur de Commerce»

1) L'instance:
Trouver le chemin/circuit le plus court entre n villes d'un réseau donné, qui passe par chaque ville.
(On n a qu'à «analyser toutes les solutions».)

2) La question:
Existe–t–il un circuit, passant exactement une fois par chacune des villes, de longueur inférieure à une constante k ?

Exemple: «Voyageur de Commerce»

n=6, k=30:



Si la réponse est **oui**, c'est très simple (polynomial !) de convaincre qqn de ce fait. Si la réponse est **non**, il faut qu'on teste tous les chemins pour être sûr.

Solution Partielle: «Backtracking»

... est une solution qui «commence bien», mais une fois qu'elle ne marche plus, il faut ***aller en arrière*** (!) quelques pas et continuer avec un autre chemin.

... à vérifier avec le voyageur de commerce

Si l'on suppose l'existence d'une ***pièce magique*** qui fait toujours le bon choix, on peut l'utiliser dans les algorithme pour atteindre des solutions polynomiales.

On appelle ***non détérministiques*** ces algorithmes, parce qu'ils peuvent toujours deviner la bonne continuation.

La Classe NP

Souvent, on ne sait pas dire si un problème décidable admet une solution polynomiale ou non.

Néanmoins, pour ces problèmes, on a souvent des *certificats*, c–à–d, des preuves polynomiales pour les cas positifs.

NP est la classe de ces problèmes. Ils permettent des solutions *nondétérministique–polynomiales*.

Evidemment, **P** est alors la classe des problèmes qui permettent des solutions *polynomiales*.

P = NP ?

Une transformation polynomiale permet de réduire un problème B à un autre problème A (en temps polynomial): elle convertit toute *instance positive* de B en une *instance positive* de A. On écrit: $B \leq_p A$.

Par conséquent, B n'est pas plus difficile que A.

Un problème A est **NP–difficile**, si pour tout $B \in$ **NP**: $B \leq_p A$.
Un problème A est **NP–complet**, si $A \in$ **NP** et est **NP**–difficile.

Soit A **NP–complet**. Alors: $A \in$ **P** ssi **P** = **NP**.

P = NP ?

La réponse n'est pas connu ...
... malgré des recherches intensives.

L'opinion partagée est:

- soit **P** $\subset$ **NP**
- soit **P = NP** n'est pas décidable

Approche descendante

On résoud un problème par une analyse descendante,
c–à–d, ***du plus général au plus spécifique***,
à l'aide *des blocs imbriqués*, representant des *algorithmes*.

Exemple: Tri par Insertion (0)

1				1
3				2
4				3
2				4
6				5
5				6

Exemple: Tri par Insertion (1)

tri insertion
entrée: un tableau d'entiers
sortie: un tableau d'entiers trié (mêmes éléments!)

chercher un élément mal placé

tant que: *un tel élément existe*

chercher sa bonne place
déplacer l'élément vers sa bonne place
chercher un élément mal placé

fin tant que

Exemple: Tri par Insertion (2)

mal placé
entrée: un tableau d'entiers
sortie: l'indice du premier élément mal placé

pour: *chaque élément du tableau, à partir du 2ième*

comparer cet élément au précedent
si: *il est plus petit, alors il est mal placée*
retourner *l'indice de cet élément;* **terminer**
fin si

fin pour
retourner *MAX, la taille du tableau;* **terminer**

Exemple: Tri par Insertion (3)

bonne place

entrée: un tableau **`tab`** d'entiers, l'indice **`pos`** du mal placé
sortie: l'indice **`newpos`** de la bonne place

`pour`: *chaque élément du tableau, de* 0 *à* **`pos`**

```
    si: l'élement est plus grand que tab[pos]
        retourner l'indice de cet élément; terminer
    fin si
fin pour
```


Exemple: Tri par Insertion (4)

déplacer

entrée: un tableau d'entiers, les indices **`pos`** et **`newpos`**
sortie: le tableau dans lequel l'élément a été déplacé
(*L'ordre des autres éléments est conservé !*)

mémoriser l'élément d'indice **`pos`** *dans* **`tmp`**
`pour`: *chaque élément du tableau, entre* **`pos-1`** *et* **`newpos`**

    copier cet élément à la position suivante

`fin pour`
copier l'élément mémorisé à la position **`newpos`**

Diviser pour Résoudre (Divide and Conquer)

Principe: (vers la récursion)

P(X):

si X est suffisamment simple, applique T(X),
sinon:

> décomposer X en instances plus petites: $X_1,...,X_n$
> pour tout X_i $(1 \leq i \leq n)$:
> résoudre $P(X_i)$ pour obtenir la solution Y_i
> recombiner les Y_i pour la solution globale Y

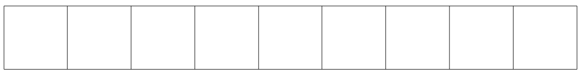
T(X): un «algorithme» qui est souvent trivial !

Diviser pour Résoudre

But: Recherche d'un élément x dans un ensemble E
Entrée: x, E (sous la forme d'un tableau trié)
Sortie: *vrai* ou *faux*

Idée: si E est complex (plus d'un élément)

- *diviser* E en trois parties (2 ensembles, 1 élément)
- *comparer* x avec l'élément de la division
- terminer, si l'élément est déjà trouvé
- ou: *poursuivre la recherche avec la moitié approprié*



Avantage: la complexité est O(log n) selon la taille n de E.