Théorie de la complexité

Cours 4

Maud MARCHAL

20 avril 2011

Plan

Plan

Complexité de problèmes

L'étude de la complexité des problèmes s'attache à **déterminer la complexité intrinsèque d'un problème** et **à classifier les problèmes selon celle-ci**. Elle apporte des réponses à des questions comme :

- Quelle est la complexité minimale d'un algorithme résolvant tel problème ?
- Existe-t-il un algorithme polynômial pour résoudre un problème donné ?
- Comment peut-on dire qu'un algorithme est optimal (en complexité) ?
- Comment peut-on montrer qu'il n'existe pas d'algorithme polynômial pour un problème ?
- Qu'est-ce qu'un problème "dur" ?
- Comment prouver qu'un problème est au moins aussi "dur" qu'un autre ?

Complexité d'un problème : définition

Définition : La complexité d'un problème est la complexité minimale dans le pire des cas d'un algorithme qui le résout.

- On s'intéresse souvent à la complexité en temps mais d'autres mesures peuvent également être utilisées (complexité en espace par exemple).
- Cette définition -un peu floue- ne précise pas quel modèle d'algorithme on choisit. L'analyse de la complexité sera différente si on programme en C ou en machine de Turing. D'où l'avantage des classes de complexité indépendantes du modèle.

Complexité d'un problème : exemple

Si on dit que la complexité d'un problème est quadratique, cela veut dire :

- qu'il existe un algorithme quadratique qui le résout ;
- ET que tout algorithme qui le résout sera au moins quadratique.

Complexité d'un problème

Calculer la complexité d'un problème est en général une chose extrêmement ardue (cf. première partie du cours). On se contente donc souvent d'encadrer :

- pour trouver une borne supérieure, il suffit de trouver **UN** algorithme ;
- pour trouver une "bonne" borne inférieure, les choses sont souvent plus dures... Ainsi, pour montrer qu'un problème est de complexité au moins exponentielle, il faut montrer que **TOUT** algorithme le résolvant est exponentiel.

Rappel : notations sur les ordres de grandeur

Soient f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$:

- $f \in O(g)$ ssi $\exists c \in \mathbb{R}^{+*}, \exists A$ tels que $\forall n > A, f(n) \leq c.g(n)$
  On dit que f est dominée asymptotiquement par g. On notera souvent $f = O(g)$.
- $f \in \Theta(g)$ ssi $\exists c, C \in \mathbb{R}^{+*}, \exists A$ tels que
  $\forall n > A, C.g(n) \leq f(n) \leq c.g(n)$
  On dit que f et g sont de même ordre de grandeur asymptotique. On notera souvent $f = \Theta(g)$.
- $f \in o(g)$ ssi $\forall \epsilon \in \mathbb{R}^{+*}, \exists A$ tels que $\forall n > 1, f(n) \leq \epsilon.g(n)$
  On dit que f est négligeable asymptotiquement devant g. On notera souvent $f = o(g)$.

Ordres de grandeur : exemples

- $n^3 + 3n + 7 \in \Theta(n^3)$
- $5n^3 + 3n + 7 \in \Theta(n^3)$
- $5n^3 + 3n + 7 \in O(n^3)$
- $5n^3 + 3n + 7 \in o(n^4)$
- $log(n) \in o(n)$

Ordres de grandeur : exemples

Exemples de temps d'exécution en fonction de la taille des données et de la complexité de l'algorithme, si on suppose que le temps d'exécution d'une instruction est de l'ordre de la microseconde :

	$log(n)$	n	$nlog(n)$	n^2	2^n
10	$3\mu s$	$10\mu s$	$30\mu s$	$100\mu s$	$1000\mu s$
100	$7\mu s$	$100\mu s$	$700\mu s$	$1/100s$	10^{14} siècles
1000	$10\mu s$	$1000\mu s$	$1/100s$	$1s$	astronomique
10000	$13\mu s$	$1/100s$	$1/7s$	$1,7min$	astronomique
100000	$17\mu s$	$1/10s$	$2s$	$2,8h$	astronomique

Trois méthodes pour trouver une borne inférieure

1. Les méthodes dites d'oracle ou d'adversaire ;
2. Les arbres de décision ;
3. Les Réductions.

Les méthodes dites d'oracle ou d'adversaire

- **Principe :** on suppose qu'il existe un algorithme utilisant moins d'un certain nombre d'opérations d'un certain type ; on construit alors une donnée qui met en défaut l'algorithme.
- Exercice : montrer que rechercher le minimum dans une liste de n éléments nécessite n-1 comparaisons.

Les arbres de décision

- **Contexte :** Ils sont utilisés pour les algorithmes de type recherche ou tri "par comparaisons" : on suppose que seules des comparaisons entre les éléments sont utilisées pour obtenir de l'information sur l'ordre ou l'égalité des éléments.
- **Description :** Un arbre de décision représente toutes les comparaisons exécutées par l'algorithme sur les données :
  - Un noeud correspond à une comparaison ;
  - Ses fils aux différentes réponses possibles de la comparaison ;
  - Une exécution possible correspond donc à une branche ;
  - Deux données sur la même branche correspondront à la même suite d'instructions.
- Exercice : montrer que tout tri par comparaisons de n éléments effectue au moins de l'ordre de n log(n) comparaisons.

Les réductions

- **Principe :** Supposons qu'on sache qu'un problème P_1 est de complexité exponentielle, donc que tout algorithme le résolvant est au moins exponentiel. Si on arrive à réduire de façon peu coûteuse le problème P_1 dans le problème P_2, P_2 sera lui aussi de complexité au moins exponentielle.
- Exercice : montrer que s'il existait un algorithme en $O(nlog(n))$ pour calculer le carré d'un entier de n chiffres, il existerait un algorithme en $O(nlog(n))$ pour calculer le produit de deux entier de n chiffres.

Plan

Problèmes d'optimisation

- Un problème d'optimisation consiste à chercher le minimum/maximum s^* d'une application f sur un **ensemble fini** S :

$$f(s^*) = min_{s \in S} f(s)$$

- Exemple : Art galery problem
  - Dans un musée, on a un ensemble de caméras disponibles. Chaque caméra peut surveiller un sous-ensemble de couloirs. L'objectif est de trouver l'ensemble minimal de caméras pour surveiller tous les couloirs.

Problèmes d'existence

- Parfois, on s'intéresse à trouver des **solutions faisables** qui satisfont un critère (et qui ne sont pas minimales) :

$$s : f(s) < L$$

- Un problème d'existence consiste à chercher un élément s d'une application f sur un **ensemble fini** S qui correspond à une propriété P.
- Les problèmes d'existence peuvent être envisagés comme des problèmes d'optimisation :

$$f : S \rightarrow \{0, 1\}$$

et $f(s) = 0$ si P est vérifiée. On cherche alors :

$$s : f(s) = 0$$

Relations entre problèmes d'optimisation et d'existence

- Un problème d'optimisation (en {0,1}) peut être associé à chaque problème d'existence.
- Inversement, un problème d'existence peut être associé à chaque problème d'optimisation. Au lieu de chercher la valeur minimale, on cherche une solution qui est inférieure à une valeur donnée.

Conséquences :

- Si le problème d'existence associé à une optimisation est difficile, alors le problème d'optimisation l'est aussi.
- Si le problème d'existence peut être résolu facilement par un algorithme A, alors il peut être utilisé (par dichotomie sur L) pour trouver l'optimum.

Plan

Introduction

- Théoriquement, les problèmes d'optimisation et d'existence peuvent toujours être résolus par une énumération exhaustive : on calcule la valeur à chaque $s \in S$ et on retient la solution de valeur minimale (ou qui correspond à la propriété P).
- Si la cardinalité de S est importante, le temps de calcul est intolérable.
- Exemples :
  - Plus court chemin dans un graphe valué par des valeurs positives : on connaît des solutions calculées en temps polynômial.
  - Stable maximal d'un graphe : pas de solution connue en temps polynômial.

Bons et mauvais algorithmes

- Un algorithme qui donne une solution (optimale) en utilisant un **temps de calcul polynômial** est acceptable (praticable).
- Un algorithme qui utilise un temps exponentiel est intolérable pour des tailles réelles quelconques. En effet :
  - Une exponentielle dépasse tout polynôme pour n assez grand ;
  - L'ensemble des polynômes est fermé : la combinaison des polynômes donne des polynômes.
  - On peut construire des algorithmes polynomiaux à partir d'algorithmes polynomiaux plus petits.
- Remarque : un problème facile peut être résolu en utilisant un mauvais algorithme (ex : plus court chemin avec énumération exhaustive).

Introduction aux classes de problèmes

Questions :

- Existe-t-il une classes de problèmes sans solution polynômiale ?
- ou bien, on ne connaît pas les bonnes solutions pour ces problèmes ?

Aujourd'hui :

- Il existe une classes de problèmes intrinsèquement difficiles.
- Les problèmes difficiles sont liés (une solution polynômiale pour un d'entre eux implique des solutions polynômiales aux autres).
- Exemple : problème du voyageur de commerce, problème de Steiner, pavage optimal, etc.

Les différentes classes

- On ne traite que des problèmes d'existence (Si un problème d'existence a une solution efficace, alors le problème d'optimisation associé a aussi une solution efficace, par dichotomie).
- Les différentes classes de problèmes :
  - Classe P
  - Classe NP
  - Les autres classes

La classe P

- **Définition** : La classe P (ou PTIME) correspond à la classe des problèmes d'existence (et d'optimisation associés) pour lesquels il existe un algorithme de résolution polynômial en temps.
  - Exemples : "être pair", "être trié" pour une liste d'entiers, "être premier" (depuis quelques années seulement), etc.
- **Définition 2 :** Un problème d'existence est dit **praticable** s'il est dans P, impraticable sinon.
  - La question "Existe-t-il un algorithme praticable pour ce problème ?" correspond donc à la question "Ce problème est-il dans P ?"

Quelques autres exemples de classes

- **PSPACE** : la classe des problèmes pour lesquels il existe un algorithme de résolution polynômial en espace. PTIME est inclus dans PSPACE (un algorithme polynômial en temps consomme au plus un espace polynômial).
- **EXPTIME** : la classe des problèmes pour lesquels il existe un algorithme de résolution exponentiel en temps.

La classe NP

- **Définition** : La classe NP regroupe les problèmes d'existence pour lesquels une proposition de solution "oui" est **vérifiable** en temps polynômial.
- La classe NP :
  1. Elle contient beaucoup de problèmes courants : problèmes de tournées, d'emploi du temps, rotation d'équipages, etc.
  2. La classe NP contient P et est contenue dans EXPTIME (et même dans PSPACE). Pour beaucoup de pronlèmes NP, on n'a pas trouvé d'algorithme polynômial, mais pour **aucun d'entre eux**, on a prouvé qu'ils ne pouvaient pas être décidés en temps polynômial.

La conjecture $P \neq NP$

1. Il existe des problèmes dans NP qui ne peuvent pas être résolus en temps P (aujourd'hui) ;
2. Il existe des problèmes dans NP pour lesquels :
   - personne n'a réussi à trouver un algorithme polynômial ;
   - personne n'a réussi à prouver la NP-complétude ;

On suppose donc que $P \neq NP$ mais personne n'a su prouver cette conjecture émise en 1971 et déclarée comme un des problèmes du millénaire par l'institut Clay qui propose 1 million de dollars pour sa résolution !

Exemple 1 : le 3-coloriage de graphes

- **Problème** : soit $G = (V, E)$ un graphe.
  On aimerait que le graphe soit coloré en 3 couleurs, i.e. trouver :
  une application col de V dans $\{1, 2, 3\}$ telle que :
  $col(v) \neq col(v')$ si v et v' sont reliés par un arc.

Exemple 2 : circuit hamiltonien

- **Problème** : soit $G = (V, E)$ un graphe.
  On cherche à savoir si le graphe contient un circuit hamiltonien, i.e. trouver :
  une application injective $sommet$ de $\{1, 2 \cdots, n\}$ dans V telle que :
  - elle soit injective (on ne passe pas deux fois par le même sommet) ;
  - $(sommet(i), sommet(i+1)) \in V$ pour $1 \leq i < n$,
  - $(sommet(n), sommet(1)) \in V$

Exemple 3 : atteindre une cible

- **Problème** : soient $x_1 \cdots x_n$ n entiers, C un entier (cible).
  On obtient la réponse "oui" en sortie si on peut obtenir C comme somme d'un sous-ensemble des x_i, i.e. on peut trouver :
  $J \subset \{1, \cdots, n\}$ tel que $C = \sum_{i \in J} x_i$.

Exemple 4 : SAT : satisfaisabilité d'une expression booléenne

- **Problème** : soient n un entier et Φ une expression booléenne avec n variables booléennes $x_1 \cdots x_n$.

  On souhaite savoir si Φ est satisfiable, i.e. il existe une valuation v telle que $v(\Phi) = Vrai$.

- Exemples :
  - si $\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_4) \wedge (\bar{x_1} \vee \bar{x_3}) \wedge (x_1 \vee \bar{x_4})$, Φ est satisfiable : $v(x_1) = Vrai$, $v(x_3) = Faux$, etc.
  - si $\Phi = (x_1 \vee x_4) \wedge (\bar{x_1} \vee \bar{x_3}) \wedge (x_1 \vee \bar{x_4}) \wedge (\bar{x_1} \vee x_3)$, Φ n'est pas satisfiable.

Liens entre les exemples ?

Pour chacun des exemples précédents, on cherche s'il existe une solution qui vérifie une certaine contrainte :

- les candidats/solutions sont des objets pas trop "gros" (coloriage, valuation, etc.) ;
- vérifier si une solution vérifie la contrainte est "facile" ;

Ces caractéristiques représentent la spécificité des problèmes appartenant à la classe NP.

Remarque : nous n'avons pas pour autant l'algorithme efficace. Enumérer les candidats solutions amène à une solution exponentielle (ex : il y a plus de 3-coloriages d'un graphe de 200 sommets que d'atomes dans l'univers).

Prouver l'appartenance à la classe NP

- Pour prouver l'appartenance à NP, il faut :
  1. proposer un codage des solutions (**certificat**) ; il faut ensuite montrer que la taille du certificat est bornée polynômialement par la taille du problème ;
  2. proposer un algorithme qui vérifie la solution selon le codage et des données (**algorithme de vérification**) ; montrer que cet algorithme a une complexité polynômiale.

Preuve de l'appartenance à la classe NP : exemple avec le 3-coloriage

- **Certificat :** un coloriage des noeuds (sous forme d'un table). La taille du certificat est bien inférieure à celle du graphe (nombre de sommets + nombre d'arcs).
- **Algorithme de vérification :** un certificat est valide ssi aucun arc ne relie deux noeuds de même couleur :

```
boolean A(col, G){
    pour chaque arc (s,d) de G faire
      si col(s)=col(d) retourner Faux;
    retourner Vrai;
```

La complexité de l'algorithme est de l'ordre de $card(V)$ donc bien polynômiale. "Etre 3-coloriable" est donc bien un problème appartenant à la classe NP.

Preuve de l'appartenance à la classe NP : exercice

- Prouver l'appartenance des problèmes du circuit hamiltonien et de la cible à la classe NP.

La définition via le non-déterminisme

- NP = Non-Déterministe Polynomial
- **Définition alternative :** Un problème P appartient à la classe NP s'il existe un algorithme non déterministe polynômial qui décide P.
- Remarque : NP=Non-Déterministe Polynomial et non pas Non Polynomial !

Algorithmes non déterministes polynômiaux

- Un algorithme non-déterministe peut être vu comme un algorithme avec des instructions de type "choix(i,1..n)" : on choisit aléatoirement un entier dans l'intervalle $[1..n]$.
  - Remarque 1 : on peut se restreindre à n=2 ;
- Un algorithme non-déterministe A est dit polynômial s'il existe un polynôme Q tel que pour toute entrée u, **tous** les calculs de A sur u ont une longueur d'exécution bornée par $Q(|u|)$.

Définition via le non-déterminisme vs. Définition via les certificats

Les deux définitions sont équivalentes !

- **Des certificats au non-déterminisme :**
  - Un certificat correspond à une suite de choix dans l'exécution de l'algorithme non déterministe. A partir d'un algorithme non-déterministe polynômial pour vérifier le problème, on construira donc la notion de certificat qui correspond à une suite de choix.
  - L'algorithme de vérification consiste à vérifier que l'exécution de l'algorithme non déterministe correspondant à la suite de choix donnée par le certificat retourne Vrai.
- **Du non-déterminisme aux certificats :**
  - A partir d'une notion de certificat et d'algorithme de vérification, on construit un algorithme non-déterministe qui consiste d'abord à générer aléatoirement un certificat (la partie non déterministe) et ensuite à le vérifier en utilisant l'algorithme -déterministe- de vérification.

La classe NP et les autres classes

- **P est inclus dans NP** : tout problème de classe P est un problème de classe NP.
  L'algorithme de vérification est l'algorithme de décision et il n'a pas besoin de certificat : on peut prendre pour certificat le mot vide.
- NP est inclus dans EXPTIME (et même PSPACE)

Bilan intermédiaire

- Un problème de classe NP est un problème pour lequel trouver une solution est peut-être difficile mais vérifier une solution est facile.
  - **P : easy to find**
  - **NP : easy to check**
- Montrer qu'un problème est NP n'est en général qu'une étape... On cherche en général ensuite à montrer qu'il est NP-dur.
- Attention : montrer qu'un problème est NP n'est pas montrer qu'il n'est pas P !
- Attention (bis) : montrer qu'un problème R est NP ne montre pas que non R est NP !

Plan

Les réductions polynômiales

- La notion de réduction permet de traduire qu'un problème n'est pas "plus dur" qu'un autre.
- Si un premier problème se réduit en un second, le premier problème est (au moins) aussi facile que le second (si la réduction est "facile").
- On peut utiliser la réduction de 2 façons :
  - pour montrer qu'un problème est dur : si le premier est réputé dur, le second l'est aussi ;
  - pour montrer qu'un problème est facile : si le deuxième est facile, le premier l'est aussi. ?

Les réductions polynômiales : exemple

Soient 2 problèmes d'existence :

1. **Problème 1** : Soit N un nombre de participants contenant une liste de paires d'ennemis. Peut-on faire p équipes de telle sorte qu'aucune équipe ne contienne une paire d'ennemis ?
2. **Problème 2** : Soit G un graphe et k un nombre de couleurs. G est-il k-coloriable ?

Quel lien existe-t-il entre les 2 problèmes ?

Les réductions polynômiales : exemple (2)

Supposons qu'on ait un algorithme pour le problème du coloriage. On transforme alors une instance du problème 1 en une instance du problème 2 :

- Les sommets du graphe sont les personnes ;
- Il y a un arc entre 2 sommets si les personnes sont ennemies.
- k=p

On a alors : G est k-coloriable ssi on peut faire p équipes.

De plus : la construction de G se fait polynômialement en fonction de l'instance du problème 1. Donc si on a un algorithme polynômial pour le p-coloriage de graphe, on en a aussi un pour le problème d'équipes. Inversement, si on n'a pas d'algorithme polynômial pour le problème d'équipes, on n'en a pas non plus pour le problème du p-coloriage.

Remarque : La réduction polynômiale marche ici dans les 2 sens.

Résumé sur les réductions polynômiales

Pour prouver qu'un problème P_1 se réduit polynômialement en un autre P_2, il faut :

- proposer une réduction, i.e. un algorithme *red* qui à une instance I de P_1, associe une instance $red(I)$ de P_2 ;
- prouver qu'elle est correcte, i.e. qu'une instance I de P_1 est positive si et seulement si $red(I)$ est une instance positive de P_2 ;
- prouver qu'elle est polynômiale.

Plan

Classes NP-dures et NP-complètes

- **Définition : problème NP-dur :** Un problème R est dit NP-dur si tout problème NP se réduit polynômialement en R.
- **Définition : problème NP-complet :** Un problème P est dit NP-complet s'il appartient à la classe NP et à la classe NP-dure.

Pourquoi montrer qu'un problème est NP-dur ?

- Montrer qu'un problème est NP-dur justifie raisonnablement qu'on n'ait pas trouvé d'algorithme polynômial pour décider de ce problème.
- Remarque : par contre, montrer qu'un problème est NP, c'est montrer qu'il n'est pas si "dur" que ça même s'il n'appartient peut-être pas à la classe P.

Démonstration de la NP-complétude d'un problème P_1

1. Démontrer que P_1 appartient à NP (vérification en temps polynômial) ;
2. Chercher un problème P_2 connu être NP-complet qui peut être transformé polynômialement en P_1 ;
3. Sachant que P_2 est NP-Complet, tous les problèmes de NP s'y transforment polynômialement. Puis, avec la transformation polynômiale trouvée, c'est aussi vrai pour P_1.

Cas particulier : si l'on trouve un cas particulier de P_1 qui correspond à $P_2 \in NP - Complet$, la preuve est immédiate.

Quelques exemples de problèmes NP-complets

- 3-coloriage de graphe (mais le 2-coloriage est facile) ;
- Problème du sac à dos binaire, du sac à dos non fractionnable (mais le sac à dos fractionnable est P) ;
- Problème de l'existence d'un circuit hamiltonien ;
- Problème du voyageur de commerce ;
- Programmation linéaire en entiers (mais la programmation linéaire en réels est P) ;
- Problème du démineur, etc.

Plan

En pratique...

Il existe de nombreux problèmes NP-durs dans divers domaines d'applications :

- réseaux, télécommunications ;
- transports ;
- gestion de production (approvisionnement, ordonnancement, etc.) ;
- organisation (emploi du temps, équipes, etc.)
- etc.

Que faire pour résoudre un problème NP-dur ?

- Supposons qu'on ait à résoudre un problème d'optimisation de la forme "trouver une solution qui optimise une fonction objectif" et qu'on ait prouvé que ce problème soit NP-dur :

On doit donc abandonner l'idée d'avoir un algorithme polynômial qui donne à coup sûr la solution optimale (à moins de prétendre prouver que P=NP !)

Que faire pour résoudre un problème NP-dur ?

On peut alors prendre 2 directions :

- abandonner l'idée d'avoir un algorithme polynômial ;
- abandonner l'idée d'avoir un algorithme qui donne toujours la solution exacte.

Abandonner l'idée d'avoir un algorithme polynômial

Montrer qu'un problème est NP-dur ne donne des informations que sur le pire des cas

Par exemple, il existe des solveurs "efficaces" qui ont été développés ces dernières années :

- ils résolvent des instances avec un million de variables ;
- ils peuvent bloquer sur certaines instances avec quelques centaines de variables.

Plusieurs alternatives :

- essayer d'obtenir un algorithme exact non polynômial optimisé ;
- essayer des méthodes d'approximation exactes ;
- utiliser des solveurs appropriés ;

Abandonner l'idée d'avoir un algorithme polynômial dans le pire des cas

Essayer d'obtenir un algorithme exact non polynômial optimisé et ne l'appliquer que sur des données relativement petites ou faciles

- Pour certains problèmes, comme le problème du sac à dos, on a des algorithmes pseudo-polynomiaux qui auront des bons comportements sous certaines conditions.
- Un **algorithme pseudo-polynômial** est un algorithme qui est polynômial si les entiers de la données sont codés en base 1 : par exemple, si la donnée est un entier n, un algorithme en $O(n)$ n'est pas polynômial mais il est pseudo-polynômial.

Abandonner l'idée d'avoir un algorithme polynômial dans le pire des cas

Essayer des méthodes d'approximation exactes

- On peut essayer des méthodes de type "Branch and Bound" (séparation/évaluation) qui évitent de parcourir l'espace de toutes les solutions en "élaguant".
- On pourra par exemple éviter de développer une solution partielle si on sait qu'elle ne pourra donner que des solutions moins bonnes qu'une solution déjà trouvée.
- On peut éventuellement paralléliser de telles méthodes.

Abandonner l'idée d'avoir un algorithme polynômial dans le pire des cas

Utiliser des solveurs appropriés

- Beaucoup de travaux ont été effectués pour améliorer les algorithmes, en particulier autour des problèmes tels que :
  - Résolution des problèmes SAT ;
  - Programmation linéaire en entiers ;
  - Programmation par contraintes.

Abandonner l'idée d'avoir un algorithme exact

- On laisse tomber la contrainte "trouver la solution optimale" et on cherchera un algorithme polynômial qui trouve une solution, parfois optimale, souvent bonne.
- Dans ce cas, on utilise des **algorithmes d'approximation** (cf. TD) : ils fournissent toujours une solution mais pas toujours optimale.

Mon problème est dur : bonne nouvelle ! !

- Dans certains contextes, comme en cryptographie, la sécurité est parfois basée sur la dureté d'un problème.
- Par exemple, RSA repose sur le fait que décomposer un entier est a priori dur mais vérifier qu'une décomposition est correcte est facile.
  - Attention : le problème d'existence associé à la décomposition d'un entier n'a pas été montré NP-dur (il est NP).
  - Attention (bis) : la sécurité doit être assurée dans tous les cas et non pas dans le pire.

Plan

Conclusion

- Il existe différentes classes de complexité, indépendantes du modèle d'algorithme choisi.
  - On retient : "NP : easy to check" vs. "P : easy to find"
- La classe des problèmes NP-durs : contient toute la complexité de NP. Les problèmes sont aussi durs que n'importe quel problème NP.
- Pour montrer qu'un problème R est NP-dur : on cherche à réduire polynômialement un problème connu NP-dur dans R.
- On montre que le problème d'existence associé au problème d'optimisation est NP-dur et on en déduit la dureté du problème initial.