Complexité de Problèmes: les classes P et NP

AAC

Sophie Tison
Université Lille 1
Master1 Informatique

Bilan de la première partie

- Programmation Dynamique

Bilan de la première partie

- Programmation Dynamique
- Algorithmes gloutons

Bilan de la première partie

- Programmation Dynamique
- Algorithmes gloutons
- Diviser pour régner

Aujourd'hui

- Début de la deuxième partie: Complexité de problèmes

Complexité de Problèmes

L'étude de la complexité des problèmes - *computational complexity* - s'attache à déterminer la complexité intrinsèque d'un problème et à classifier les problèmes selon celle-ci.

Complexité de Problèmes: les questions qu'elle aborde

Complexité de Problèmes: les questions qu'elle aborde

- Quelle est la complexité minimale d'un algorithme résolvant un problème donné?

Complexité de Problèmes: les questions qu'elle aborde

- Quelle est la complexité minimale d'un algorithme résolvant un problème donné?
- Existe-t-il un algorithme polynomial pour résoudre un problème donné?

Complexité de Problèmes: les questions qu'elle aborde

- Quelle est la complexité minimale d'un algorithme résolvant un problème donné?
- Existe-t-il un algorithme polynomial pour résoudre un problème donné?
- Comment peut-on dire qu'un algorithme est optimal (en complexité)?

Complexité de Problèmes: les questions qu'elle aborde

- Quelle est la complexité minimale d'un algorithme résolvant un problème donné?
- Existe-t-il un algorithme polynomial 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 polynomial pour un problème?

Complexité de Problèmes: les questions qu'elle aborde

- Quelle est la complexité minimale d'un algorithme résolvant un problème donné?
- Existe-t-il un algorithme polynomial 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 polynomial pour un problème?
- Qu'est-ce qu'un problème "dur"?

Complexité de Problèmes: les questions qu'elle aborde

- Quelle est la complexité minimale d'un algorithme résolvant un problème donné?
- Existe-t-il un algorithme polynomial 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 polynomial pour un problème?
- Qu'est-ce qu'un problème "dur"?
- Comment prouver qu'un problème est au moins aussi "dur" qu'un autre?

La complexité d'un problème

Definition

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

La complexité d'un problème

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

C'est souvent la complexité en temps qu'on considère mais on peut s'intéresser à d'autres mesures comme par exemple la complexité en espace.

La complexité d'un problème

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

C'est souvent la complexité en temps qu'on considère mais on peut s'intéresser à d'autres mesures comme par exemple la complexité en espace.

Cette définition -un peu floue- ne précise pas quel modèle d'algorithme on choisit: l'analyse de la complexité si elle est fine (par exemple, être linéaire, être quadratique) sera différente si on programme en C ou "en Machine de Turing" d'où l'intérêt de classes de complexité indépendantes du modèle.

La complexité d'un problème

La complexité d'un problème

Par exemple si on dit que la complexité d'un problème est quadratique cela veut dire:

La complexité d'un problème

Par exemple si on dit que la complexité d'un problème est quadratique cela veut dire:

- qu'il existe un algorithme quadratique qui le résout

La complexité d'un problème

Par 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

La complexité d'un problème

Calculer la complexité d'un problème est ardu en général. On se contente souvent d'encadrer:

La complexité d'un problème

Calculer la complexité d'un problème est ardu en général. On se contente souvent d'encadrer:

- pour trouver une borne supérieure, il suffit de trouver **UN** algorithme.

La complexité d'un problème

Calculer la complexité d'un problème est ardu en général. On se contente 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 difficiles; pour montrer par exemple qu'un problème est de complexité au moins exponentielle, il faut montrer que **TOUT** algorithme le résolvant est exponentiel.

La complexité d'un problème

Calculer la complexité d'un problème est ardu en général. On se contente 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 difficiles; pour montrer par exemple qu'un problème est de complexité au moins exponentielle, il faut montrer que **TOUT** algorithme le résolvant est exponentiel.

Quand les deux bornes coïncident, c'est l'idéal mais c'est assez rare!

L'exemple de la multiplication

L'exemple de la multiplication

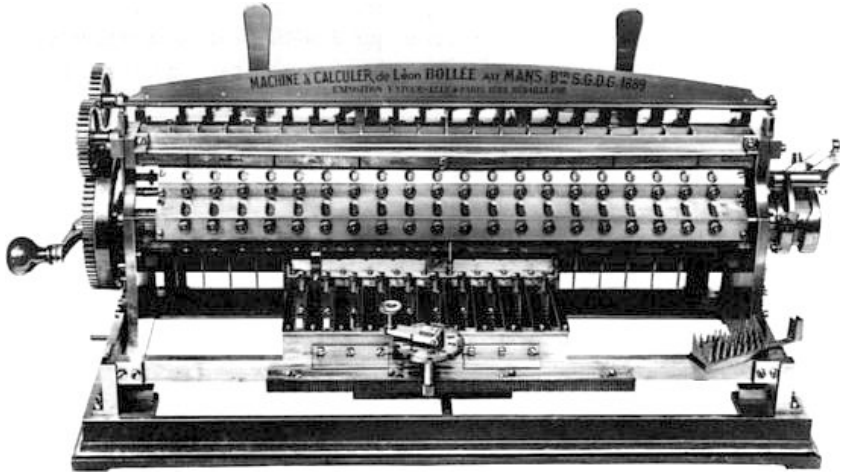
- On conjecture que la multiplication de deux entiers de longueur n peut se faire en $O(n \log n)$

L'exemple de la multiplication

- On conjecture que la multiplication de deux entiers de longueur n peut se faire en $O(n \log n)$
- Le meilleur algorithme connu est en $O(n \log n \log \log n)$.

L'exemple de la multiplication

- On conjecture que la multiplication de deux entiers de longueur n peut se faire en $O(n \log n)$
- Le meilleur algorithme connu est en $O(n \log n \log \log n)$.
- La meilleure borne inférieure trouvée est $O(n)$.



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:

Le 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.

Les méthodes dites d'oracle ou d'adversaire:

Le 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 ainsi que rechercher le maximum dans une liste de n éléments nécessite $n - 1$ comparaisons.

Les arbres de décision

Les arbres de décision

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.

Les arbres de décision

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.
Un arbre de décision "représente" toutes les comparaisons exécutées par l'algorithme sur les données d'une certaine taille:

Les arbres de décision

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.
Un arbre de décision "représente" toutes les comparaisons exécutées par l'algorithme sur les données d'une certaine taille:

- Un noeud correspond à une comparaison,

Les arbres de décision

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.
Un arbre de décision "représente" toutes les comparaisons exécutées par l'algorithme sur les données d'une certaine taille:

- Un noeud correspond à une comparaison,
- ses fils aux différentes réponses possibles de la comparaison (donc si le test est à valeur booléenne, les noeuds sont binaires);

Les arbres de décision

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.
Un arbre de décision "représente" toutes les comparaisons exécutées par l'algorithme sur les données d'une certaine taille:

- Un noeud correspond à une comparaison,
- ses fils aux différentes réponses possibles de la comparaison (donc si le test est à valeur booléenne, les noeuds sont binaires);
- Une exécution possible correspond donc à une branche

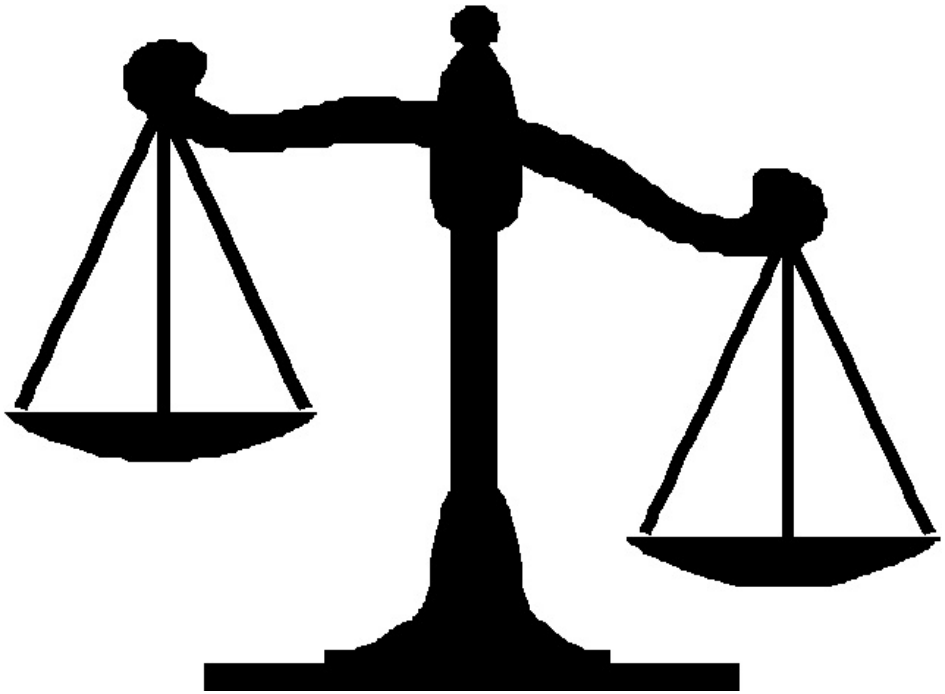
Les arbres de décision

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.
Un arbre de décision "représente" toutes les comparaisons exécutées par l'algorithme sur les données d'une certaine taille:

- Un noeud correspond à une comparaison,
- ses fils aux différentes réponses possibles de la comparaison (donc si le test est à valeur booléenne, les noeuds sont binaires);
- Une exécution possible correspond donc à une branche
- Deux données correspondant à la même branche correspondront à la même suite d'instructions.

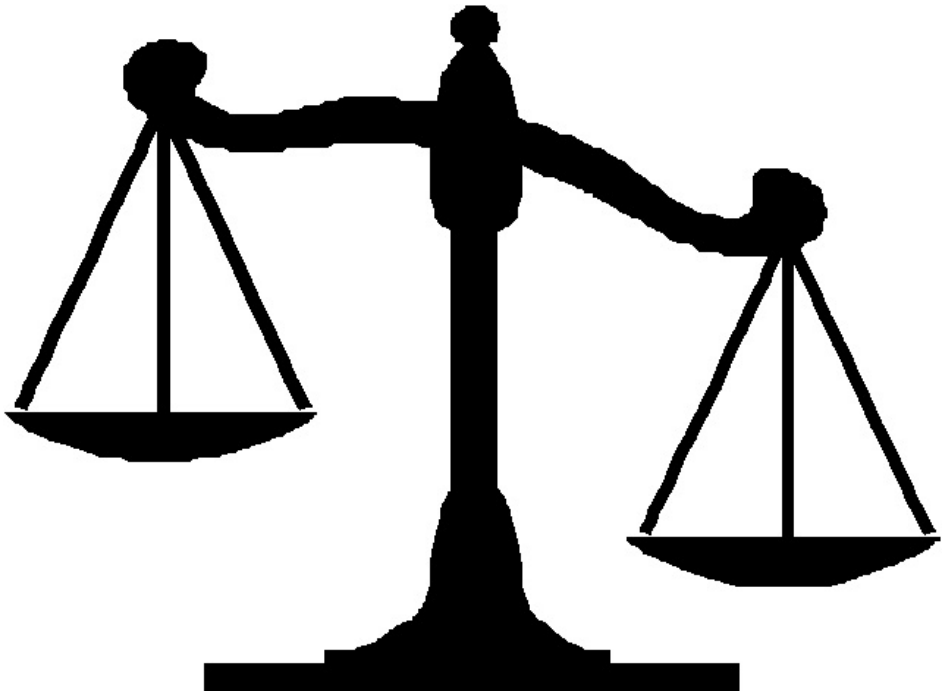
Les arbres de décision: la pesée

Soit 9 pièces de monnaies, dont une est fausse et plus légère que les autres. On possède une balance à plateaux. On cherche à trouver la fausse pièce en minimisant le nombre de pesées. Quelle méthode optimale?



Les arbres de décision: la pesée, suite

Soit 12 pièces de monnaies, dont une est fausse et soit plus légère, soit plus lourde que les autres. On possède une balance à plateaux. On cherche à trouver la fausse pièce en minimisant le nombre de pesées.
Quelle méthode optimale?



Les arbres de décision

Exercice

Montrer que tout recherche d'une valeur par "comparaison" parmi n éléments triés effectue au moins de l'ordre de $\log n$ *comparaisons dans le pire des cas.*

Les arbres de décision

Exercice

Montrer que tout tri par comparaisons de n éléments effectue au moins de l'ordre de $n \log n$ comparaisons dans le pire des cas.

Les réductions

Le principe est simple.

Les réductions

Le principe est simple.

Supposons par exemple qu'on sache que le problème *DURDUR* est de complexité exponentielle, donc que tout algorithme le résolvant est au moins exponentiel.

Les réductions

Le principe est simple.

Supposons par exemple qu'on sache que le problème *DURDUR* est de complexité exponentielle, donc que tout algorithme le résolvant est au moins exponentiel.

Si on arrive à réduire d'une façon "peu coûteuse" le problème *DURDUR* dans le problème *MONPB*, *MONPB* sera lui aussi de complexité au moins exponentielle.

Les réductions

Exercice

Montrer que si il existait un algorithme en $O(n \log n)$ pour calculer le carré d'un entier de n chiffres, il existerait un algorithme en $O(n \log n)$ pour calculer le produit de deux entiers de n chiffres,

Les classes de Problèmes

L'idée est de classifier les problèmes selon l'ordre de grandeur de leur complexité.
Tout d'abord pour simplifier, on va se limiter aux problèmes de décision ou *p*ropriétés.

Definition

Une propriété est une fonction à valeurs booléennes.

Les classes de Problèmes

L'idée est de classifier les problèmes selon l'ordre de grandeur de leur complexité.
Tout d'abord pour simplifier, on va se limiter aux problèmes de décision ou *p*ropriétés.

Definition

Une propriété est une fonction à valeurs booléennes.

Un problème de décision peut donc être vu comme un ensemble $\mathcal{I}$ d'instances du problème avec pour chaque instance une réponse "oui" ou "non".

Les classes de Problèmes

L'idée est de classifier les problèmes selon l'ordre de grandeur de leur complexité.
Tout d'abord pour simplifier, on va se limiter aux problèmes de décision ou *p*ropriétés.

Definition

Une propriété est une fonction à valeurs booléennes.

Un problème de décision peut donc être vu comme un ensemble $\mathcal{I}$ d'instances du problème avec pour chaque instance une réponse "oui" ou "non".

Exemple: "être premier" est une fonction qui à tout entier naturel associe oui -si il est premier-, non sinon.

Abstrait / Concret

A une propriété abstraite définie sur un ensemble quelconque, on peut associer une propriété concrète définie sur des mots -les représentation des instances-.

Abstrait / Concret

A une propriété abstraite définie sur un ensemble quelconque, on peut associer une propriété concrète définie sur des mots -les représentation des instances-.

Par exemple à la propriété abstraite "être premier" définie sur les entiers, on associera la propriété "être la représentation binaire d'un entier premier" définie sur $\{0,1\}^*$.

Abstrait / Concret

A une propriété abstraite définie sur un ensemble quelconque, on peut associer une propriété concrète définie sur des mots -les représentation des instances-.

Par exemple à la propriété abstraite "être premier" définie sur les entiers, on associera la propriété "être la représentation binaire d'un entier premier" définie sur $\{0,1\}^*$.

Cela suppose donc qu'on a choisi une représentation des instances du problème.

Propriété / langage:

On peut donc ensuite associer à la propriété le langage des (représentations des) données vérifiant cette propriété.

Propriété / langage:

On peut donc ensuite associer à la propriété le langage des (représentations des) données vérifiant cette propriété.

Par exemple, à la propriété "être premier", on associera l'ensemble des représentations des entiers premiers dans une certaine base non unaire.

Propriété / langage:

On peut donc ensuite associer à la propriété le langage des (représentations des) données vérifiant cette propriété.

Par exemple, à la propriété "être premier", on associera l'ensemble des représentations des entiers premiers dans une certaine base non unaire.

Donc, vérifier si un entier est premier se ramène à tester si sa représentation (=un mot) est dans le langage.

Propriété / langage

Décider une propriété se ramène donc à tester l'appartenance d'un mot à un langage.

Un premier exemple: La classe P

Definition

La classe P (ou *PTIME*) est la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynômial en temps.

Un premier exemple: La classe P

Definition
La classe P (ou *PTIME*) est la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynômial en temps.

Cette définition est (quasiment-) indépendante du modèle d'algorithme choisi: un algorithme polynômial en C sera polynomial si il est traduit en termes de machine de Turing et les modèles classiques de calcul (excepté les ordinateurs quantiques) sont polynomialement équivalents.

P EST LA CLASSE DES PROBLÈMES PRATICABLES

Un problème de décision est dit **praticable** si il est dans P, impraticable sinon.

P EST LA CLASSE DES PROBLÈMES PRATICABLES

Un problème de décision est dit **praticable** si il est dans P, impraticable sinon.

"existe-t-il ou non un algorithme praticable pour ce problème?"

P EST LA CLASSE DES PROBLÈMES PRATICABLES

Un problème de décision est dit **praticable** si il est dans P, impraticable sinon.

"existe-t-il ou non un algorithme praticable pour ce problème?"
se ramène pour le problème à

P EST LA CLASSE DES PROBLÈMES PRATICABLES

Un problème de décision est dit **praticable** si il est dans P, impraticable sinon.

"existe-t-il ou non un algorithme praticable pour ce problème?"
se ramène pour le problème à
"être ou ne pas être dans P?"

Exemples de problèmes praticables ou non

- ”Etre pair” est

Exemples de problèmes praticables ou non

- ”Etre pair” est P.
- ”Etre trié” pour une liste d’entiers est

Exemples de problèmes praticables ou non

- ”Etre pair” est P.
- ”Etre trié” pour une liste d’entiers est P.

Exemples de problèmes praticables ou non

- ”Etre pair” est P.
- ”Etre trié” pour une liste d’entiers est P.
- ”Etre premier” est

Exemples de problèmes praticables ou non

- "Etre pair" est P.
- "Etre trié" pour une liste d'entiers est P.
- "Etre premier" est P . On le sait depuis quelques années seulement.

Exemples de problèmes praticables ou non

- ”Etre pair” est P.
- ”Etre trié” pour une liste d'entiers est P.
- ”Etre premier” est P . On le sait depuis quelques années seulement.
- ” le problème des échecs généralisés -i.e. la taille de l'échiquier est non fixée, on a une configuration, on cherche à savoir si elle est gagnante-

Exemples de problèmes praticables ou non

- ”Etre pair” est P.
- ”Etre trié” pour une liste d'entiers est P.
- ”Etre premier” est P . On le sait depuis quelques années seulement.
- ” le problème des échecs généralisés -i.e. la taille de l'échiquier est non fixée, on a une configuration, on cherche à savoir si elle est gagnante- n'est pas P.

Exemples de problèmes praticables ou non

- "Etre pair" est P.
- "Etre trié" pour une liste d'entiers est P.
- "Etre premier" est P . On le sait depuis quelques années seulement.
- " le problème des échecs généralisés -i.e. la taille de l'échiquier est non fixée, on a une configuration, on cherche à savoir si elle est gagnante- n'est pas P.
- Décider si une expression rationnelle étendue (avec le complémentaire en plus de l'union, la concaténation et l'étoile) représente tous les mots

Exemples de problèmes praticables ou non

- "Etre pair" est P.
- "Etre trié" pour une liste d'entiers est P.
- "Etre premier" est P . On le sait depuis quelques années seulement.
- " le problème des échecs généralisés -i.e. la taille de l'échiquier est non fixée, on a une configuration, on cherche à savoir si elle est gagnante- n'est pas P.
- Décider si une expression rationnelle étendue (avec le complémentaire en plus de l'union, la concaténation et l'étoile) représente tous les mots n'est pas P.

Exemples de problèmes praticables ou non

- "Etre pair" est P.
- "Etre trié" pour une liste d'entiers est P.
- "Etre premier" est P . On le sait depuis quelques années seulement.
- " le problème des échecs généralisés -i.e. la taille de l'échiquier est non fixée, on a une configuration, on cherche à savoir si elle est gagnante- n'est pas P.
- Décider si une expression rationnelle étendue (avec le complémentaire en plus de l'union, la concaténation et l'étoile) représente tous les mots n'est pas P.
- On ne sait pas si décider si une expression rationnelle "normale" représente tous les mots est P (sans doute non..).

Quelques autres exemples de classes

- *PSPACE*: la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynomial en espace.

Quelques autres exemples de classes

- *PSPACE*: la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynomial en espace.

  PTIME est inclus dans *PSPACE*: un algorithme polynomial en temps "consomme au plus un espace polynomial".

Quelques autres exemples de classes

- *PSPACE*: la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynomial en espace.

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

Quelques autres exemples de classes

- *PSPACE*: la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynomial en espace.

  PTIME est inclus dans *PSPACE*: un algorithme polynomial en temps "consomme au plus un espace polynomial".
- *EXPTIME*: la classe des problèmes de décision pour lesquels il existe un algorithme de résolution exponentiel en temps.
- *NP*: A suivre...

Quelques autres exemples de classes

- *PSPACE*: la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynomial en espace.

  PTIME est inclus dans *PSPACE*: un algorithme polynomial en temps "consomme au plus un espace polynomial".
- *EXPTIME*: la classe des problèmes de décision pour lesquels il existe un algorithme de résolution exponentiel en temps.
- *NP*: A suivre...

Exercice

Expliquer pourquoi PSPACE est inclus dans EXPTIME

La classe NP

1. Elle contient beaucoup de propriétés associées à des problèmes courants: problèmes de tournées, d'emploi du temps, placement de tâches, affectation de fréquences, rotation d'équipages, découpage...

La classe NP

1. Elle contient beaucoup de propriétés associées à des problèmes courants: problèmes de tournées, d'emploi du temps, placement de tâches, affectation de fréquences, rotation d'équipages, découpage...
2. La classe *NP* contient *P* et est contenue dans *ExpTime* (et même dans *Pspace*).
   Pour beaucoup de propriétés *NP*, on n'a pas trouvé d'algorithme polynomial, mais pour **aucune d'entre elles**, on n'a prouvé qu'elle ne pouvait pas être décidée en temps polynomial.

La classe NP

1. Elle contient beaucoup de propriétés associées à des problèmes courants: problèmes de tournées, d'emploi du temps, placement de tâches, affectation de fréquences, rotation d'équipages, découpage...
2. La classe NP contient P et est contenue dans $ExpTime$ (et même dans $Pspace$).
   Pour beaucoup de propriétés NP, on n'a pas trouvé d'algorithme polynomial, mais pour **aucune d'entre elles**, on n'a prouvé qu'elle ne pouvait pas être décidée en temps polynomial.
3. on suppose que $P \neq NP$ mais personne jusque maintenant n'a su prouver -ni infirmer- cette conjecture émise en 1971 et déclarée un des problèmes du millénaire par l'Institut Clay qui propose 1 million de dollars pour sa résolution!

P vs NP Problem by Clay Institute

Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe!

P vs NP Problem by Clay Institute

Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.

Exemple 1: Le 3-coloriage de graphes

Donnée: $G = (S, A)$ un graphe

Sortie: oui, si le graphe peut être coloré en 3 couleurs i.e. on peut trouver:

une application col de S dans $\{1, 2, 3\}$ telle que

$col(s) \neq col(s')$ si s, s' sont reliés par un arc ($(s, s') \in A$ ou $(s', s) \in A$)

Exemple 2: Le circuit Hamiltonien

Donnée: $G = (S, A)$ un graphe, n un entier Sortie: oui, si le graphe contient un circuit hamiltonien i.e. on peut trouver:

une application injective *sommet* de $\{1, 2, ..., n\}$ dans S telle que:

elle soit injective (on ne passe pas deux fois par le même sommet)

$(sommet(i), sommet(i+1)) \in A$ pour $1 <= i < n$

$(sommet(n), sommet(1)) \in A$

Exemple 3: Atteindre la cible

Donnée: $x_1, ..., x_n$, n entiers
c un entier (cible)

Sortie: oui, si on peut obtenir c comme somme d'un sous-ensemble des x_i

i.e. on peut trouver

$J \subset \{1, ..., n\}$ tel que $c = \sum_{i \in J} x_i$

Exemple 4: SAT: satisfiabilité d'une expression Booléenne

Donnée: n, un entier, et Φ une expression booléenne avec n variables booléennes, $x_1, ..., x_n$

Sortie: oui, si Φ est satisfiable, i.e. il existe une valuation v telle que $v(\Phi) = Vrai$

Exemples:
si $\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_4) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4)$

Φ est satisfiable: $v(x_1) = Vrai, v(x_3) = Faux, ...$

si $\Phi = (x_1 \vee x_4) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4) \wedge (\neg x_1 \vee x_3)$

Φ n'est pas satisfiable (Comment le prouver?)

QUEL EST LE LIEN?

Pour chacune des propriétés précédentes, on cherche si il existe une solution qui vérifie une certaine contrainte.

.les ”candidats-solutions” sont des objets pas trop ”gros”: un coloriage, une valuation..

.vérifier si une solution vérifie la contrainte est ”facile”

C’est la spécificité des *NP* ...

Remarque: ça ne nous donne pas pour autant d’algo efficace ... Enumérer les candidats solutions nous amène à une solution exponentielle! Par exemple il y a plus de 3-coloriages d’un graphe de 200 sommets que d’atomes dans l’univers.


C'EST
TOUT
POUR
AUJOURD'
HUI.