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

- 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

La complexité d'un problème

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 "fine" de la complexité (par exemple, être linéaire, être quadratique) sera différente selon le modèle d'où l'intérêt des 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:

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

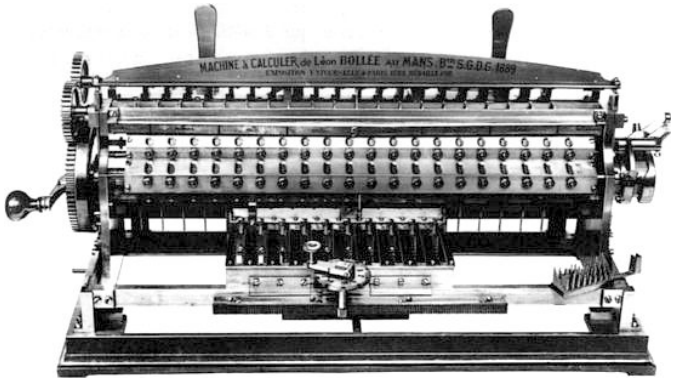
La complexité d'un problème

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

- 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



La multiplication

pour deux entiers de longueur n, combien d'opérations élémentaires sur les chiffres fait-on?

- L'algorithme de l'école primaire est en $\Theta(n^2)$.
- Karatsuba est le premier à avoir amélioré cette ordre de grandeur par un algorithme (diviser pour régner) en $O(n^{\log_2 3})$.
- Le meilleur algorithme connu (Schönhage- Strassen), basé sur la FFT, est en $O(n \log n \log\log n)$.
- On conjecture(?) qu'il y a un algorithme en $O(n \log n)$
- La meilleure borne inférieure trouvée est $O(n)$.

La factorisation d'un entier de longueur n

Donnée: un entier x dont la représentation binaire est de longueur n
Sortie: sa factorisation en nombres premiers
Donnée: un entier x dont la représentation binaire est de longueur n , un entier $m < n$
Sortie: oui si x a un facteur premier de taille au plus m.

- Tester tous les facteurs jusque $\sqrt{x}$: il y en a environ $2^{n/2}$.
- Le meilleur algorithme connu est le crible général de corps de nombres (GNFS) : il n'est pas polynomial.
- factoriser un nombre de 232 chiffres, en utilisant des centaines de machines, a pris deux ans. Certains entiers du challenge RSA de 2007 ne sont toujours pas factorisés.
- Certains protocoles de cryptographie reposent sur la difficulté de ce problème.
- Il existe un algorithme quantique polynomial pour la factorisation (algorithme de Schor).

Trois méthodes pour trouver une borne inférieure

1. Les méthodes dites d'adversaire
2. Les **arbres de décision**
3. Les **Réductions**

L'exemple du maximum

Combien de comparaisons nécessite la recherche du maximum dans une liste de n éléments?

Déterminer le gagnant d'un tournoi?

Dans un tournoi, chaque joueur sauf le vainqueur a perdu une fois.

Donc, il faut au moins $n - 1$ matchs!

Rechercher le maximum dans une liste de n éléments nécessite $n - 1$ comparaisons.

De même, tout élément différent du maximum doit avoir perdu une fois dans une comparaison

Sinon, un élément qui n'a pas perdu et n'a pas été déclaré le maximum; l'adversaire le remplace par une valeur plus grande que le maximum: l'algorithme donnera le même maximum qu'au départ, ce qui n'est pas possible.

$n - 1$ comparaisons sont nécessaires et suffisantes pour la recherche du maximum d'une liste de n élements. .

La méthode dite d'adversaire

Le principe: On suppose qu'il existe un algorithme utilisant moins d'un certain nombre d'opérations d'un certain type; l'adversaire ou oracle construit alors une donnée qui met en défaut l'algorithme.

le maximum ET le minimum

Exemple

Qu'en est-il si l'on veut rechercher le maximum ET le minimum dans une liste de n éléments?

On peut trouver un algorithme qui utilise environ 3 n/2 comparaisons.

On peut montrer par la méthode d'adversaire que tout algorithme fera au moins 3 n/2 -2 comparaisons dans le pire des cas.

La recherche dans une liste triée

Donnée: un tableau de n éléments triés, une valeur x
On cherche à déterminer si x est dans le tableau. On peut "uniquement" comparer x avec une valeur du tableau.

Combien de comparaisons au moins doit-on faire dans le pire des cas?

Une recherche dichotomique fait environ $\log n$ comparaisons dans le pire des cas. Peut-on faire mieux?

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.

La recherche dichotomique

Input: T trié croissant non vide, x
Output: retourne Vrai Ssi x présent dans T
boolean rechDico(int x)
int $g = 0, d = T.length - 1$;
while $(g < d)$ **do**
  // Invariant: T trié, $g \leq d$, si x est dans T $T[g] \leq x \leq T[d]$
  int $m = (g + d)/2$;
  if $(T[m] < x)$ **then**
    $g = m + 1$;
  else
    $d = m$;
  end
end
return (T[g]==x);

Sur un tableau de 8 éléments, l'arbre de décision sera de hauteur 3 (voir tableau).

Les arbres de décision

La recherche dans une liste triée

Tout algorithme de 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.

En effet, le nombre de feuilles de l'arbre est au moins n, la hauteur de l'arbre est donc de l'ordre de $\log n$.

La recherche dichotomique est donc en un certain sens optimale.

Trier une liste

Quelle est la complexité optimale d'un algorithme de tri par comparaisons?
On connaît des algorithmes en $\Theta(n \log n)$.

Les tris par comparaisons

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

Le nombre de feuilles de l'arbre est au moins $n!$.

Formule de Stirling: $n! \sim \sqrt{2\pi n}\ \left(\frac{n}{e}\right)^n$

$\log n!$ est de l'ordre de $n \log n$.

Attention: ne s'applique pas aux tris comme le tri par paquets ou le tri par base qui ne sont pas basés sur des comparaisons et supposent des hypothèses sur les données.

Calculer le carré d'un nombre

Donnée: x un entier de longueur n
Sortie: x^2

- Combien d'opérations élémentaires sur les chiffres doit-on faire au moins?
- Est-ce plus ou moins "dur" que le problème de la multiplication?

- Dans un sens, on peut appliquer l'algorithme de la multiplication pour élever au carré!
- Dans "l'autre sens", si on avait un algorithme en $n \log n$ pour le carré, on en aurait un pour la multiplication car:
  $a * b = ((a+b)^2 - a^2 - b^2)/2)$
  On a réduit le problème de multiplication dans le problème de l'élévation au carré!

Les réductions

Le principe est simple: coder un problème en un autre.
On peut alors utiliser ce qu'on sait de la complexité d'un problème pour en déduire une borne inférieure sur la complexité d'un autre problème.

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.

Trois méthodes pour trouver une borne inférieure

1. Les méthodes dites d'adversaire
2. Les **arbres de décision**
3. Les **Réductions**

Les classes de Problèmes

L'idée est de classifier les problèmes en grandes familles ou classes selon l'ordre de grandeur de leur complexité, en essayant d'obtenir si possible des classes indépendantes du modèle de calcul.

Les Propriétés

Pour simplifier, on va se limiter aux problèmes de décision ou propriétés.

Propriétés

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.

Les Propriétés

Pour simplifier, on va se limiter aux problèmes de décision ou propriétés.

Propriétés

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.

Pourquoi se limiter aux propriétés?

- C'est plus simple à étudier.
- On peut souvent associer à un problème, une propriété de complexité équivalente

L'exemple de la factorisation d'entiers

- Factoriser
  - Donnée: un entier n
  - Sortie: sa factorisation en facteurs premiers
- Existence d'un facteur de taille m
  - Donnée: un entier n, un entier m, $1 < m < n$
  - Sortie: oui, ssi x a un facteur premier inférieur ou égal à m.

Si on a un algorithme polynomial pour le deuxième, on en a un pour le premier (Pourquoi?).

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 ce langage.

Propriété/langage

Décider une propriété se ramène donc à tester l'appartenance d'un mot à un langage, celui des représentations des instances positives de la propriété (les données qui vérifient la propriété).

Un premier exemple: La classe P

La classe P

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

Cette définition est (quasiment-) indépendante du modèle d'algorithme choisi: un algorithme polynomial 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.

"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 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 cherche à savoir si une configuration, est gagnante

Pourquoi généraliser?

Pour le jeu des échecs, un échiquier a 64 cases (8*8): le nombre de configurations, si il est immense est fini.
C'est le cas pour la plupart des jeux usuels: l'espace des configurations est fini.

En complexité algorithmique , on étudie la complexité asymptotique.
Donc on généralise les jeux pour étudier leur complexité algorithmique, par exemple pour les échecs à des échiquiers $n * n$ en adaptant les règles.

Cela donne un bon indice de la complexité du cas particulier du jeu usuel.

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 cherche à savoir si une configuration, 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.
  tous les mots" est P (sans doute non).