Informatique théorique

Nhan LE THANH
DUT informatique
janvier 2007

Plan du cours

1- Structures usuelles
2- Raisonnement par récurrence et dénombrement
3- Raisonnement par induction et suites récurrentes
4- Logique proportionnelle et prédicative
5- fonction récursives et Machine Turing
6- Calculabilité et complexité
7- Classe NP

Séance 6: de la calculabilité à la complexité

Plan

1- Décidabilité et indécidabilité
2- Complexité algorithmique

VI.1 Décidabilité : Histoire

- Intuitivement, P est décidable s'il existe un algorithme qui pour chaque x dit "OUI" ou "NON" à la question : "Est-ce que P(x) est vrai ?
- *équations diophantiennes*
  - On appelle équation diophantienne une équation du genre P(x, y, z, ...) = 0, où P est un polynôme à coefficients entiers
  - Hilbert posait la question (1900): "existe-t-il un procédé mécanique permettant de dire, après un nombre fini d'étapes, si une équation diophantienne donnée possède des solutions entières?".
  - De fait, on ne disposait pas, jusqu'à une date récente (Andrew Whiles, 1993) d'une telle démonstration.

VI.1 Décidabilité : Histoire

- Hilbert posait la question générale de savoir si l'on pouvait trouver un procédé mécanisable capable de résoudre toutes les questions mathématiques récalcitrantes.
- C'est à partir de cette dernière question que Turing s'est mis au travail. Elle est plus générale que la première question:
  - en effet, on pourrait imaginer qu'il n'y ait pas de procédure uniforme pour résoudre tous les problèmes mathématiques
  - mais qu'il y en ait une pour résoudre la problème particulier des équations diophantiennes.
- En fait, ni l'une ni l'autre n'existe.

VI.1 Décidabilité : Histoire

- En logique mathématique, le terme **décidabilité** recouvre deux concepts liés :
  - la décidabilité *logique* et
  - la décidabilité *algorithmique*.
- L'indécidabilité est la négation de la décidabilité.
- Dans les deux cas il s'agit de formaliser l'idée qu'on ne peut pas toujours conclure lorsque l'on se pose une question, même si celle-ci est sous forme logique.

VI.1 Décidabilité : logique

- Une proposition (on dit aussi énoncé) est dite **décidable** dans une théorie axiomatique, si on peut la démontrer ou démontrer sa négation dans le cadre de cette théorie.
- Un énoncé mathématique est donc indécidable dans une théorie s'il est impossible de le déduire, ou de déduire sa négation, à partir des axiomes.
- Pour distinguer cette notion d'indécidabilité de la notion d'indécidabilité algorithmique (voir ci-dessous), on dit aussi que l'énoncé est *indépendant* du système d'axiomes.

VI.1 Décidabilité : logique

- En logique classique, d'après le théorème de complétude, une proposition est indécidable dans une théorie s'il existe des modèles de la théorie où la proposition est fausse et des modèles où elle est vraie.
- Si l'on admet le fait assez intuitif que la géométrie euclidienne est cohérente — la négation de l'axiome des parallèles ne se déduit pas des autres axiomes — l'axiome des parallèles est bien alors indépendant des autres axiomes de la géométrie, ou encore indécidable dans le système formé des axiomes restant.

VI.1 Décidabilité : logique

- Une théorie mathématique pour laquelle tout énoncé est décidable est dite complète, sinon elle est dite incomplète.
- **1er Théorème d'incomplétude de Gödel :** Dans n'importe quelle théorie récursivement axiomatisable, cohérente et capable de « formaliser l'arithmétique », on peut construire un énoncé arithmétique qui ne peut être ni prouvé ni réfuté dans cette théorie.
- **2e Théorème d'incomplétude de Gödel :** Si T est une théorie cohérente qui satisfait des hypothèses analogues, la cohérence de T, qui peut s'exprimer dans la théorie T, n'est pas démontrable dans T.

VI.1 Décidabilité : algorithmique

- Un problème de décision est dit décidable s'il existe un *algorithme*, une *procédure mécanique* qui termine en un nombre fini d'étapes, qui le décide, c'est-à-dire qui réponde par oui ou par non à la question posée par le problème.
- S'il n'existe pas de tels algorithmes, le problème est dit indécidable.
- Le problème décidable algorithmique réfère à la notion de *calculabilité* en cherchant un algorithme qui s'arrête à un temps fini et fournit la réponse OUI ou NON à la question posée.

VI.1 Décidabilité : algorithmique exemples

- Tous les sous-ensembles finis des entiers sont décidables (il suffit de tester l'égalité à chacun des entiers de l'ensemble).
- On peut construire un algorithme pour décider si un entier naturel est pair ou non (on fait la division par deux, si le reste est zéro, le nombre est pair, si le reste est un, il ne l'est pas), donc l'ensemble des entiers naturels pairs est décidable ;
- il en est de même de l'ensemble des nombres premiers.
- Le problème de savoir si une proposition est vraie dans l'arithmétique de Presburger, c'est-à-dire dans la théorie des nombres entiers naturels avec l'addition mais sans la multiplication, est décidable.

VI.1 Indécidabilité : algorithmique exemples

- Le problème de l'arrêt.
  - Dans ce cas, les questions portent sur tous les programmes informatiques dans un langage suffisamment puissant
  - Il s'agit de savoir si oui ou non un ordinateur s'arrêtera lorsqu'il exécute un programme à partir de l'état initial de la mémoire.
  - L'indécidabilité du problème de l'arrêt a été prouvée par Alan Turing.
- La question de savoir si oui ou non un énoncé de la logique du premier ordre est universellement valide (démontrable dans toute théorie), dépend de la signature du langage choisie (les symboles d'opération ou de relation ...).
  - Ce problème, parfois appelé problème de la décision, est indécidable pour le langage de l'arithmétique, et plus généralement pour n'importe quel langage égalitaire du premier ordre qui contient au moins un symbole de relation binaire (comme < ou ≤)

- Le programme suivant en langage C (supposons que p n'intervienne pas ailleurs), déterminer si l'on doit libérer le bloc mémoire si tôt après son allocation est équivalent à décider le problème de l'arrêt sur f(). Il est donc indécidable !

```
char *p = malloc(1000);
f();
*p = 1;
```


VI.2 Complexité: Motivation

- Un algorithme est une procédure finie et mécanique de résolution d'un problème.
- Exemples : les algorithmes d'Euclide, l'algorithme de Dijkstra ...
  - Un algorithme doit se terminer sur toutes les données possibles du problème et doit fournir une solution correcte dans chaque cas.
  - Pour résoudre informatiquement un problème donné, on implante donc un algorithme sur un ordinateur.
- Mais, pour un problème donné, il existe bien souvent plusieurs
- algorithmes.
- Y a-t-il un intérêt à choisir ? et si oui comment choisir ?
  - En pratique, il n'est même pas suffisant d'avoir un algorithme.
- Il existe des problèmes pour lesquels on a des algorithmes, mais qui restent «!informatiquement non résolus!». C'est parce que les temps d'exécution sont vite exorbitants.
- On cherche alors des heuristiques pour abaisser ces temps de calcul.

VI.2 Complexité: Introduction

- Le mot complexité recouvre en fait deux réalités :
  - la complexité des algorithmes :
    - C'est l'étude de l'efficacité comparée des algorithmes.
    - On mesure ainsi le temps et aussi l'espace nécessaire à un algorithme pour résoudre un problème.
    - Cela peut se faire de façon expérimentale ou formelle.
  - la complexité des problèmes
    - La complexité des algorithmes a abouti à une classification des problèmes en fonction des performances des meilleurs algorithmes connus qui les résolvent.
    - Techniquement, les ordinateurs progressent de jour en jour. Cela ne change rien à la classification précédente.
    - Elle a été conçue indépendamment des caractéristiques techniques des ordinateurs.
- Les domaines connexes de la complexité et de la preuve de programmes utilisent toutes les notions vues jusqu'à présent dans ce cours.

VI.2 Complexité: Temps d'exécution

- On implante un algorithme dans un langage de haut niveau pour résoudre un problème donné.
- Le temps d'exécution du programme dépend :
  - des données du problème pour cette exécution
  - de la qualité du code engendré par le compilateur
  - de la nature et de la rapidité des instructions offertes par l'ordinateur (facteur 1 à 1000)
  - de l'efficacité de l'algorithme
  - …
  - et aussi de la qualité de la programmation !
- A priori, on ne peut pas mesurer le temps de calcul sur toutes les entrées possibles. Il faut trouver une autre méthode d'évaluation.
- L'idée est de s'affranchir des considérations subjectives (programmeur, matériel…).
- On cherche une grandeur n pour «!quantifier!» les entrées.

VI.2 Complexité: Modèles de calcul

- Les machines de Turing (TM) ou les Random Access Machines(RAM) sont des machines abstraites. Elles servent d'«étalon» à la mesure des complexités en temps et en espace des fonctions calculables.
  - Thèse de Church-Turing (indépendamment mais les 2 en 1936)
  - Toute fonction effectivement calculable l'est par une machine de Turing.
- Notre optique ici n'est évidemment pas de présenter cette
- théorie dans son entier, mais juste d'en appréhender les
- principes les plus simples.
- Notre approche pragmatique nous servira surtout à comparer des algorithmes résolvant un même problème et à décider rigoureusement lesquels sont les meilleurs.

VI.2 Complexité: Evaluation des coûts en séquentiel

- Dans un programme strictement séquentiel,les «boucles» sont disjointes ou emboîtées. Ceci exclut toute récursivité.
- Les temps d'exécution s'additionnent :
  - choix : (si C alors J sinon K) T(n) = TC(n) + max { TJ(n), TK(n) }
  - itération bornée (pour i de j à k faire B)
    T(n) = (k-j+1).(Tentête(n) + TB(n)) + Tentête(n))

  entête est mis pour l'affectation de l'indice de boucle et le test de continuation.
- Le cas des boucles imbriquées se déduit de cette formule.
  - itération non bornée (tant que C faire B)
    T(n) = #boucles . (TB(n) + TC(n)) + TC(n)
    (répéter B jusqu'à C)
    T(n) = #boucles . (TB(n) + TC(n))
    Le nombre de boucles #boucles s'évalue inductivement.
  - appel de procédures : il n'y a pas d'appel récursif on peut ordonner les procédures de sorte que la ième ait sa complexité qui ne dépend que des

VI.2 Complexité: Evaluation des coûts en récursif

- On sait que les algorithmes du type «!diviser pour régner!» donnent lieu à des programmes récursifs.
- Comment évaluer leur coût ? Il faut trouver :
  - la relation de récurrence associée
  - la base de la récurrence, en examinant les cas d'arrêts de la récursion
  - et aussi la solution de cette équation
- Techniques utilisées : résolution des équations de récurrence ou de partition classique ou par les séries génératrices.
- Exemples
  - On avait évalué le temps T(n) de la recherche dichotomique dans un tableau de n cases à T(n) = 1 + log2(n)
  - La complexité en temps pour résoudre le problème des tours de Hanoï était T(n) = 2n - 1

VI.2 Complexité: Variantes du temps d'exécution

- Temps d'exécution dans le pire des cas
  Tpire(n) = $\max_n$ { T(n) }
  - Exemple :
    le cas le pire pour le tri rapide (quicksort) est quand le tableau est déjà trié, le cas le pire pour le tri par sélection est quand le tableau est trié dans l'ordre inverse.
- Temps d'exécution en moyenne
  Tmoy(n) = Sn P(n) . T(n)
  - avec P(n) une loi de probabilité sur les entrées. Fréquemment, on utilise la loi de probabilité uniforme. T(n) peut aussi être une valeur moyenne.

  Tmoy&unif(n) = (1/#n) Sn T(n)
- Temps d'exécution dans le meilleur des cas
  Tmeil(n) = minn { T(n) }

VI.2 Complexité: Exemple (extrait du tri par sélection)

```
1. min ← i
2. pour j de i+1 à n faire
3. si A[j] < A[min] alors
4. min ← j
```

→1 affectation
→ 1 aff.+ 1 comp. + (1 aff.+ 1 comp.).#boucle
→ 1 comparaison . #boucles
→ (si test vrai : 1 affectation) . #boucles

- Dans le pire des cas, quand la table est triée par ordre inverse : 4 (n-i) + 3
- Supposons que, sur les (n-i) tests, la (petite) moitié est évaluée à vrai. En moyenne, 4 [n-i] + 3 [n-i] + 3
- Les notations précédentes, appelées plancher et plafond donnent respectivement la moitié de l'argument arrondie à l'entier inférieur et la moitié de l'argument arrondie à l'entier supérieur.
- Dans le meilleur des cas, quand le tableau est trié, on n'exécute jamais l'instruction min ← j. 3 (n-i+1)

VI.2 Complexité: Estimation asymptotique

- En complexité, on ne considère que des fonctions positives de IN dans IR. On ne veut pas évaluer précisément les temps d'exécution (d'autant que ça dépend des machines…). On se contente d'estimations.
- On dit que f positive est asymptotiquement majorée (ou dominée) par g et on écrit f = O (g)
- quand il existe une constante c, c>0, telle que pour un n
- assez grand, on a : f(n) ≤ c g(n)
- On définit symétriquement la minoration de f par g : f = Ω(g).
- On dit que f est du même ordre de grandeur que g et on écrit f = Θ(g) quand
  f = O(g) et g = O(f)
  autrement dit
  $c1\ g(n) \leq f(n) \leq c2\ g(n)$

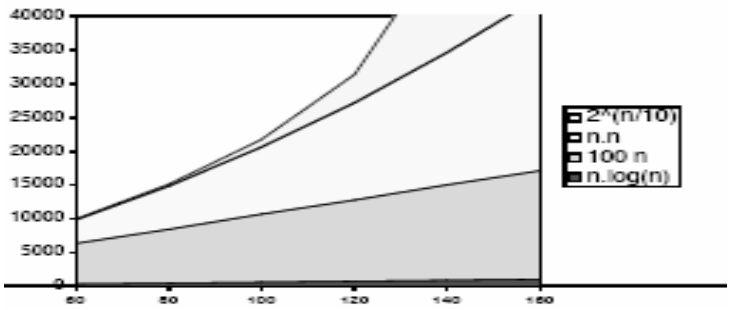
VI.2 Complexité: Ordres de grandeurs

- du plus petit au plus gran: O(1), O(log(n)), O(nx), O(n), O(n.log(n)), O(nc), O(cn), O(n!) avec $O < x < 1 < c$
- la notation f = O(g) est scabreuse car elle ne dénote pas une égalité mais plutôt l'appartenance de f à la classe des fonctions en O(g).
- Exemple : $4n^2+n = O(n^2)$ et $n^2-3 = O(n^2)$
  sans que pour n assez grand on ait $4n^2+n = n^2-3$
- un algorithme est polynomial s'il existe un entier p tel que son coût est $O(n^p)$.

O(1)	constant
O(log(n))	logarithmique
O(n)	linéaire
O(n log(n))	n log(n)
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel

VI.2 Complexité: Illustration

- En pratique,
  - un algorithme à une complexité exponentielle est inutilisable.
  - pour n pas trop grand, les algorithmes polynomiaux sont encore efficaces.



VI.2 Complexité: **Propriétés**

- Réflexivité
  - g = O(g) donc O(g) = O(O(g))
  - g = q(g) donc q(g) = q(q (g))
- Symétrie
  - f = q(g) ssi g = q(f)
- Transitivité
  - f = O (g) et g = O(h) alors f = O(h)
  - f = q(g) et g = q(h) alors f = q(h)
- Produit par un scalaire
  - l>0, l . O(g) = O(g)
- Somme et produit de fonctions
  - O(f) + O(g)) = O(max (f,g))
  - O(f) . O(g)) = O(f.g)

VI.2 Complexité: **Exemples**

- Voici le temps de calcul d'algorithmes récursifs classiques :
- recherche dichotomique dans un tableau de n cases $T(n) = 1 + \log2(n)$
  soit $T(n) = O(\log_2(n))$
- problème des tours de Hanoï
  $T(n) = 2^n - 1$
  soit $T(n) = O(2^n)$
- factorielle : temps linéaire
- évaluation de polynôme (Horner) : linéaire sur le degré
- produit de matrice/vecteur : temps quadratique.
- tri rapide (quicksort) : n.log(n)

VI.2 Complexité: **Exemples**

- 1. fonction fusion(L,K:liste):liste → O (1)
- 2. **si** L vide **alors** → O (1)
- 3. résultat ← K → O (1)
- **4. sinon si** K vide **alors** → O (1)
- 5. résultat ← L → O (1)
- 6. **sinon si** elmCourant(L)< elmCourant(K) **alors** → O (1)
- 7. résultat ← concatène(elmCourant(L),fusion(suivant(L),K)) → O(1)+O((T(n-1))
- **8. sinon**
- **9.** résultat ← concatène(elmCourant(K),fusion(L,suivant(K)) → O(1)+O(T(n-1))
- Si n=0, on exécute les lignes 1,2,3 : temps O(1)
- Si n=1, on exécute les lignes 1,2,4,5 : temps O(1)
- Si n>1, on exécute 1,2,4,6,7 ou 1,2,4,6,8,9 : temps O(1) + T(n-1)
- Soit à résoudre l'équation T(0) = a et T(n) = b + T(n-1).
- ...
- La résolution donne un coût linéaire : T(n) = n.b + a = O(n)

VI.2 Complexité: **Exemples**

- 1. fonction split(L:liste): un couple de listes → O (1)
- 2. J,K ← listeVide → O (1)
- 3. **si** vide(L) **alors** → O (1)
- 4. résultat ← (L,L) → O (1)
- **5. sinon si** vide(suivant(L)) **alors** → O (1)
- 6. résultat ← (L, listeVide) → O (1)
- 7. **sinon**
- **8.** i ← 1 → O (1)
- 9. **tant que** non listeVide(L) **faire** → O (1)
- 10. e = retire(L) → O (1)
- 11. **si** impair(i) **alors** → O (1)
- 12. ajoute (e,J) → O (1)
- 13. **sinon**
- 14. ajoute (e,K) → O (1)
- 15. **finsi**
- 16. i ← i + 1 → O (1)
- 17. **fin tant que**
- 18. résultat ← (J,K) → O (1)
- **19. finsi**

- Si n=0 ou 1 : temps O(1)
- Si n>1, on exécute la boucle «!tant que!» n fois : temps O(n)
- La fonction split a donc un coût linéaire.

VI.2 Complexité: **Exemples**

- 1. procédure triFusion(L:liste): liste → O (1)
- 2. J,K ← listeVide → O (1)
- 3. **si** vide(L) **alors** → O (1)
- 4. résultat ← L → O (1)
- **5. sinon si** vide(suivant(L)) **alors** → O (1)
- 6. résultat ← L → O (1)
- 7. **sinon**
- **8.** (J,K) = split(L) → O(n)
- 9. J ← triFusion(J) → O(T(n/2))
- 10. K ← triFusion(K) → O(T(n/2))
- 11. résultat ← fusion(J,K) → O(n)
- 12. **finsi**

- on suppose que la taille de la liste est une puissance entière de 2,
- sans quoi ça complique et on ne peut envisager la résolution par équations de partition.
- si n=0 ou 1 : lignes 1,2,3,4 ou 1,2,3,5,6 : temps O(1)
- si n>1 avec n = 2k : on exécute les lignes 1,2,3,5,7,8,9,10,11. En
- particulier, il y a deux appels à des fonctions (8,11) et deux appels récursifs (9,10)
- On pose l'équation : T(1) = a et n>1, T(n) = 2 T(n/2) + b.n
- La résolution donne T(n) = a.n+ b.n.log2(n) donc le tri Fusion a un coût en O(n.log(n)).

VI.2 Complexité: **Classes**

- **1/ Classe L (LOGSPACE)**
  Un problème de décision qui peut être résolu par un algorithme déterministe en espace *logarithmique* par rapport à la taille de l'instance est dans L.
- **2/ Classe NL**
  Cette classe s'apparente à la précédente mais pour un algorithme non-déterministe.
- **3/ Classe P**
  Un problème de décision est dans *P* s'il peut être décidé par un algorithme déterministe en un temps *polynomial* par rapport à la taille de l'instance. On qualifie alors le problème de polynomial.
- **4/ Classe NP**
  Un problème NP est *N*on-déterministe *P*olynomial (et non pas *Non polynomial*, erreur très courante).
- **5/ Classe Co-NP (Complémentaire de NP)**
  C'est le nom parfois donné pour l'équivalent de la classe *NP*, mais avec la réponse *non*.
- **6/ Classe PSPACE**
  Elle regroupe les problèmes décidables par un algorithme déterministe en espace polynomial par rapport à la taille de son instance.
- **7/ Classe NSPACE ou NPSPACE**
  Elle réunit les problèmes décidables par un algorithme non-déterministe en espace polynomial par rapport à la taille de son instance.
- **8/ Classe EXPTIME**
  Elle rassemble les problèmes décidables par un algorithme déterministe en temps exponentiel par rapport à la taille de son instance.
- **9/ Classe NEXPTIME**
  Elle rassemble les problèmes décidables par un algorithme non déterministe en temps exponentiel par rapport à la taille de son instance.

VI.2 Complexité: relation entre Classes

- Liaison entre temps et entre espace
  - P ⊆ NP et symétriquement P ⊆ co-NP
  - L ⊆ NL ⊆ PSPACE = NPSPACE et symétriquement Co-NP ⊆ PSPACE
  - P ⊆ NP ⊆ EXPTIME ⊆ NEXPTIME
- Liaison entre temps et espace
  - Si M fonctionne en temps t(n) alors M fonctionne en espace au plus t(n))
- Complexité en temps et espace
  - P ⊆ NP ⊆ PSPACE = NPSPACE et symétriquement Co-NP ⊆ PSPACE
  - P ⊆ NP⊆ PSPACE = NPSPACE ⊆ EXPTIME ⊆ NEXPTIME