Notes de Cours.

Les théories de l'informatique : calculabilité et complexité

Préambule : ce cours est inspiré en partie du cours donné par J.Betrema à Bordeaux.

Deux questions fondamentales ont toujours intrigué les informaticiens. La première concerne ce qui est calculable par un ordinateur (donc par une procédure automatique) et ce qui ne l'est pas ?

C'est la question à laquelle s'attaque **la théorie de la calculabilité.**

A partir de 1935 des réponses scientifiques ont été fournies à cette question (Alonzo Church père du λ-calcul, Kurt Gödel qui a montré qu'il existe des propositions vraies indémontrables en arithmétique, Alan Turing avec ses travaux sur le déchiffrement …).

Thèse de Church. La réponse à la question est indépendante de la technologie. Tous les modèles de calcul raisonnables (langage C par exemple) utilisés en pratique sont équivalents au sens de la calculabilité donc en terme de puissance de calcul. Ainsi ce qui est calculable dans un modèle peut-être traduit (compilé) dans un autre.

λ-calcul ≈ machine de Turing
≈ langage de programmation raisonnable
≈ langage C

Dans la suite du cours nous utiliserons donc le langage C comme modèle de calcul (en supposant que les entiers sont non bornés).

Nous verrons que certains problèmes n'ont pas de solution algorithmique. Par exemple : Peut-on déterminer à la compilation si un programme quelconque va boucler ou non ? La réponse à cette question est non.

Plus récemment (autour de 1970) c'est posé une autre question dont l'impact pratique est encore plus important : qu'est-ce qui est calculable efficacement et qu'est-ce qui ne l'est pas ?

La théorie de la complexité essaie de traiter cette question.

Essayons d'abord de formaliser la notion "efficacement".

Soit le problème de la partition
Données : un ensemble de n objets et chaque objet a un poids. Peut-on partitionner cet ensemble en 2 sous-ensembles tels que la somme des poids des objets de chaque sous-ensemble soit égale.

Prenons n=100. Il y 2^{99} (resp. 2^{n-1}) partitions possibles. Un ordinateur qui fonctionne à 2 GHz depuis le début de la création de l'Univers aurait exécuté moins de 2^{90} instructions. Un algorithme en 2^n n'est donc pas exécutable en pratique. Ainsi un algorithme qui essaie toutes les combinaisons possibles serait voué à l'échec d'un point de vue pratique.

Par convention on admet qu'un problème n'est pas soluble en pratique s'il n'existe pas d'algorithme pour le résoudre qui s'exécute en temps polynomial par rapport à la taille du problème (la taille du problème

est le nombre de bits pour stocker la donnée). Nous dirons qu'un tel problème est difficile. Sinon s'il existe un algorithme qui s'exécute en temps polynomial par rapport à la taille du problème nous dirons que le problème est facile.

Cette convention est raisonnable, elle traduit le fait qu'un algorithme polynomial permet de multiplier la taille de la donnée traitée par unité de temps si la puissance de l'ordinateur est multipliée. Ainsi un algorithme en $\theta(n^3)$ permet de multiplier par 2 la taille des problèmes traités en 1s si la puissance de l'ordinateur est multipliée par 8 alors qu'avec un algorithme en $\theta(2^n)$ la taille de la donnée traitable en 1s augmente simplement 3.

Résumé.

Etant donné un problème, la théorie de la calculabilité cherche à savoir s'il existe un algorithme pour le résoudre (le problème est-il décidable (ou calculable) ou indécidable). S'il est décidable, la théorie de la complexité cherche à savoir s'il existe un algorithme efficace pour le résoudre (le problème est-il facile ou difficile).

Le concept de réduction :

La réduction est l'outil de base pour déduire si un problème est décidable ou pas, si un problème a un algorithme efficace ou pas pour le résoudre.

Soient P et Q, deux problèmes. Supposons que ProcQ résout le problème Q et que l'on puisse écrire procP qui résout P de la façon suivante :

```
procP(D) :
    début
    …..
    procQ(f(D))
    …...
    end
```

Les ….. représentent une partie de la procédure autre que l'appel à procQ alors on a les résultats suivants :

Q calculable $\Rightarrow$ P calculable (si évidemment f est calculable et tout ce qui est extérieur à l'appel à procQ est calculable). En effet il suffit d'avoir une procédure qui résout Q pour avoir une procédure qui résout P.

Q facile $\Rightarrow$ P facile (si évidemment f est calculable en temps polynomial, si le nombre d'appels à procQ est polynomial et tout ce qui est extérieur à cet appel s'exécute en temps polynomial). En effet une procédure avec les bonnes propriétés pour Q (exécution en temps polynomial) entraîne la connaissance d'une procédure avec les bonnes propriétés pour P.

Attention ne pas faire le raisonnement inverse et faux : Comme pour résoudre P on utilise une procédure qui résout Q alors P est plus "complexe" que Q (procP n'est pas forcément la meilleure procédure pour résoudre P).

De même sous les mêmes hypothèses

P non calculable ⇒ Q non calculable (si Q était calculable, on aurait une procédure pour P ce qui est faux par hypothèse).

P difficile ⇒ Q difficile (un algorithme polynomial pour Q entraînerait un algorithme polynomial pour P).

Quelques rappels sur le calcul de la complexité d'un algorithme :

Cycle :
Données : G=(V,E), un graphe non orienté.
Résultat Un booléen égal à true si et seulement si G possède un cycle.

Ce problème peut-être résolu en 0(n).

Prenons l'arbre couvrant de poids minimum
Données : G=(V,E), poids : V → N
(i) On trie les arêtes dans ordre de poids croissant : e_1, e_2, … e_m
(ii) G'=(V,E') avec E'=∅.
i=1 ; cpt=0 ;
tant que cpt<n-1 faire
si non cycle(G'+e_i) alors E'=E' {e_i} ; cpt++ ; finsi ;
i++ ;
fin tq ;

(i) et (ii) se font en séquence, donc la complexité de l'algorithme est la somme des complexités de (i) et (ii).

Complexité de (i) : S'il y a m arêtes, le tri a une complexité en O(m log(m)) au pire en utilisant un bon tri comme le tri par tas ou le tri fusion. On peut exprimer cette complexité en fonction de n le nombre de sommets, on obtient alors du O(n^2log(n)).
.
Complexité de (ii) : On a une boucle donc pour obtenir la complexité on somme la complexité de chaque itération. Lors d'une itération de la boucle on regarde si un graphe possède un cycle. Comme le graphe a moins de n sommets la complexité est en O(n). Donc la complexité de (ii) est en O(pn) où p est le nombre d'itérations de la boucle lors de l'exécution (donc p est plus petit que m).

La complexité globale est donc polynomiale.

Remarque : on peut faire beaucoup mieux pour implémenter l'algorithme de Kruskal.

Théorie de la Calculabilité.

Chapitre 1. Numérotations

Le premier principe sur lequel repose la théorie de la calculabilité est le suivant :
Tout objet informatique est représentable (codable) par un entier naturel.

Tout objet informatique peut-être codé par un entier, ce qui permettra de ne considérer que des fonctions de N dans N.

On peut coder différents ensembles d'objets pour cela il suffit de trouver une fonction de l'ensemble d'objets dans N (en général bijective). Il faut également que cette fonction soit calculable et que l'inverse le soit aussi. Voici différents exemples :

Entiers relatifs

x	0	-1	1	-2	-x	x
c(x)	0	1	2	3	-2x-1	2x

Codage : $x>0$ $c(x)=2x$
$x<0$ $c(x)=-2x-1$

Décodage si z est impair alors decode(z)=-(z+1)/2
sinon decode(z) =z/2

C'est bien une bijection (de plus sa programmation et la programmation de la fonction inverse est facile). N et Z ont la même cardinalité. On remarque que le programme
z=0 ; while (1) {ecrire(decode(z) ; z++;} permet d'écrire tous les entiers relatifs (dans le sens où tout entier relatif sera écrit au bout d'un temps fini). Cet ensemble est donc énumérable (par un programme).

Couples d'entiers

On trie les couples par (x+y) croissant puis par ordre lexicographique en cas d'égalité.

(x,y)	(0,0)	(1,0)	(0,1)	(2,0)	(1,1)	(0,2)	(3,0)	(2,1)	(1,2)	….
x+y	0	1	1	2	2	2	3	3	3	
z=rang(x,y)	0	1	2	3	4	5	6	7	8	

Soit rang(x, y)=(x+y)(x+y+1)/2 + y

Exercice : écrire le programme qui permet de calculer x et y à partir du code et d'énumérer tous les couples (x,y).

Solution : soit z le code d'un couple on calcule t le plus grand entier tel que t(t+1) est inférieur à 2z. y est alors égal à z-t(t+1)/2 et x est égal à t-y. Ainsi 39 code le couple (5,3) (t est égal à 8, donc y=3 et x=5).

On remarquera qu'écrire les couples par ordre lexicographique ne permet pas de les écrire tous (le couple (1,0) ne sera jamais écrit). Tout ordre total n'induit donc pas forcément un codage.

Généralisation au codage de triplets (ou de nuplets)

On peut donc coder les triplets par h(x,y,z) = rang(rang(x,y),z) et généraliser aux n-nuplets.

Listes d'entiers

A chaque liste *u* on associe un nombre $\sigma(u)$ égal à la somme des entiers additionnée de la longueur de la liste. Si deux listes ont la même valeur pour σ on les met dans l'ordre lexicographique.

Mots (de longueur finie) sur un alphabet fini.

Les plus courts d'abord. A égalité de longueur on utilise l'ordre alphabétique

Remarque : l'ordre alphabétique seul ne suffit pas (il y a une infinité de mots commençant par A). La stratégie ne marche pas sur un alphabet de taille infini (il y a une infinité de mots de longueur 1).

Exercice : trouver une stratégie pour écrire tous les mots même si l'alphabet est infini (mais dénombrable).

Procédures C.

int p (int x) { …. ; return ….. ;}

Les procédures pouvent être considérées comme un texte (donc un mot sur un alphabet fini on ajoutant un séparateur ou des séparateurs de mots à l'alphabet). On peut générer tous les textes et ne sélectionner que les textes qui peuvent représenter une fonction C (contrôler à l'aide d'un analyseur syntaxique). On les numérote à l'aide d'un compteur que l'on incrémente à chaque rencontre d'un texte représentant une fonction C valide.

Les procédures sont donc (effectivement) énumérables : on peut trouver un algorithme qui génère la $n^{ième}$ procédure pour tout entier n. On peut donc écrire une procédure universelle qui permet de simuler l'exécution de n'importe quelle procédure sur n'importe quelle donnée (dans un couple (x,y) d'entiers on peut supposer que x est le code de la fonction et y le code de la donnée).

Conclusion :
Dans ce chapitre, on a vu des ensembles dénombrables (mis en bijection avec N). Ces ensembles ont aussi la propriété d'être énumérables par un algorithme (dans le sens où tout élément de l'ensemble est affiché au bout d'un temps fini). En fait on verra que de nombreux ensembles dénombrables (en bijection avec N) ne sont pas énumérables par un algorithme : par exemple l'ensemble des procédures C qui ne bouclent sur aucune donnée.

Chapitre 2. Paradoxes de la diagonalisation et problème de l'arrêt

1. La preuve de Cantor.

Une fonction f est dite totale si elle est définie pour toute valeur (pour tout x, f(x) a une valeur).

Théorème de Cantor. L'ensemble des fonctions totales de N dans N n'est pas dénombrable.
Preuve. Suposons que l'on puisse numéroter les fonctions totales de N dans N : f_0, f_1, f_2 …f_i … et définissons la fonction $\gamma(n)=f_n(n)+1$. Cette fonction est bien une fonction totale de N dans N donc par hypothèse il existe f_m telle que $\gamma=f_m$. Que vaut $\gamma(m)$?
$f_m(m)$ par hypothèse et $f_m(m)+1$ par définition. D'où la contradiction.

2. Les fonctions calculables.

Si p est une procédure, n un entier : p(n) désigne le résultat de la procédure p sur la donnée n. Ce résultat est un entier si l'exécution se termine et il est indéfini sinon.

A une procédure p est donc associée une fonction mathématique.

Une fonction f est dite calculable s'il existe une procédure qui la calcule.

D'après Cantor il existe des fonctions non calculables car l'ensemble des procédures est dénombrable contrairement à l'ensemble des fonctions totales (a fortiori des fonctions).

Reprenons l'argument de cantor
Soient p_0, p_1, p_2 … p_n … les procédures et définissons γ par $\gamma(n)=p_n(n)+1$.

Cette fonction γ est calculable car à partir de n on peut calculer p_n et l'exécuter sur la donnée n. γ peut donc être calculée par une procédure et il existe donc i tel que $\forall x, \gamma(x)=p_i(x)$. Que vaut $\gamma(i)$?
$p_i(i)$ par hypothèse ou $p_i(i)+1$ par définition.

La seule possibilité est que $\gamma(i)$ n'est pas défini et donc que γ n'est pas une fonction totale.

Conclusion :
Dans tout modèle raisonnable de calcul, il existe des fonctions calculables strictement partielles.

3. Fonctions calculables totales

On ne peut pas les énumérées. En effet refaisons le même raisonnement : soit
q_0, q_1, q_2 … q_n … les fonctions calculables totales. La fonction $\gamma(n)=q_n(n)+1$ est totale mais elle ne peut être calculable (on reproduit le même raisonnement par l'absurde que précédemment).

Soit T l'ensemble des procédures qui calculent des fonctions totales. T est dénombrable. Il existe donc des bijections de T dans N mais aucune de ces bijections n'est calculable.

4. Indécidabilité du problème de l’arrêt

problème de l’arrêt : $h(p,x) = 1$ si $p(x)$ est défini et 0 sinon

définissons la fonction suivante $\gamma(n) = p_n(n)+1$ si $h(p_n,n)=1$ et 0 sinon

Remarquons que les 2 fonctions (h et γ) sont totales.

Si h est calculable alors il est facile de voir que γ l’est aussi. Donc il existe p_i telle que $\gamma=p_i$. Dans ce cas il est facile de voir que $\gamma(i)$ par hypothèse est égale à $p_i(i)$ et par définition à $p_i(i)+1$.

Cette preuve par l’absurde montre que h n’est pas calculable.

On dira de façon équivalente que le problème de l'arrêt d'un programme sur une donnée est indécidable.

Application. Soit une suite quelconque d’ensembles D_i. Construire un ensemble qui n’appartient pas à cette suite.

Chapitre 3. Récursion et Le théorème de Rice

1. Le principe de récursion

Le chapitre précédent a examiné en détail le paradoxe de la diagonalisation et notamment la preuve que le théorème de l'arrêt d'un programme sur une donnée est indécidable (il n'existe pas de programme qui prend en entrée un programme p et une donnée x et rend comme résultat true si p s'arrête sur x et false si p ne s'arrête pas sur la donnée x) .

L'usage de la récursion donne une autre preuve. Supposons que h soit calculable par une procédure hp. Soit gamma la procédure suivante

```
int gamma (int x) {
      if (hp (gamma,x)) while (1) ;
      else return 0 ;
}
```

Si gamma(x) est défini on en déduit que gamma(x) boucle et si gamma(x) est non défini que gamma(x)=0. Donc par l'absurde il n'existe pas de procédure qui calcule h.

2. Le théorème de Rice

Pourquoi limiter la méthode précédente au problème de l'arrêt ?
Soit $\delta(p) = 1$ si $\forall x$, $p(x)=x$ et 0 sinon. Autrement dit $\delta(p) = 1$ ssi p calcule l'identité.
Si δ est calculable par une procédure delta alors on peut écrire :

```
int gamma (int x) {
      if (delta (gamma)) return 0 ;
      else return x ;
}
```

On en déduit
Si $\forall x$ gamma(x)=x $\Rightarrow \delta$(gamma)=1 $\Rightarrow \forall x$ gamma(x)=0
Si $\exists x$ gamma(x)$\neq$x$\Rightarrow \delta$(gamma)=0$\Rightarrow \forall x$ gamma(x)=x

Dans tous les cas on arrive donc à une contradiction qui implique que delta n'existe pas.

On peut donc facilement généraliser :
Définition : un prédicat P est trivial si il vaut toujours la même valeur quelque soit ses paramêtres (i.e. $\forall x$ P(x)=1 ou $\forall x$ P(x)=0).

Théorème de Rice. Soit P un prédicat tel que

$$\forall x\ p(x)=q(x) \Rightarrow P(p)=P(q)$$

alors P est indécidable s'il n'est pas trivial.

Preuve.

Rappel : un prédicat est une fonction totale booléenne.
P non trivial donc il existe p_0 et p_1 tels que $P(p_0)=0$ et $P(p_1)=1$.
Supposons qu'il existe procP qui calcule P alors on peut écrire

```
int gamma (int x) {
      if (procP (gamma)) return  p0(x) ;
      else return p1(x) ;
}
```

Que vaut P(gamma) ? Toutes les hypothèses mènent à une absurdité.

Remarque : L'indécidabilité du problème de l'arrêt est une bonne nouvelle pour les informaticiens. Ecrire des programmes corrects qui calculent ce que l'on attend d'eux sera toujours une activité nécessitant de la réflexion et de l'expérience.

Application :

Soit p une procédure. Montrez que savoir si $\forall x, p(x)< x+1$ est un problème indécidable.
Soit q une procédure. Montrer que savoir si pour tout x, q(x) est pair est indécidable

0n donne à chaque fois deux preuves différentes, l'une utilisant le théorème de Rice et l'autre le principe de récursion.

Chapitre 4. Ensemble récursivement énumérable

1. **Définitions.**

La fonction caractéristique X d'un ensemble d'entiers E est la fonction définie par
X(x)=1 si x∈ E et 0 sinon.

E est dit décidable si sa fonction caractéristique est calculable.
E est récursivement énumérable si E={f(0), f(1), f(2), ….} où f désigne une fonction calculable totale. On dit que f énumère les éléments de E (énumérable par un algorithme qui permet de donner tous les éléments).

Théorème. Tout ensemble fini est décidable
Preuve. On peut le représenter par sa liste d'éléments et donc la fonction caractéristique est calculable par le parcours de la liste.

Théorème. Tout ensemble décidable est récursivement énumérable.
Preuve. Soit fc qui calcule la fonction caractéristique de E. Voici une procédure qui énumère les éléments de E :

```
int enum (int n) {
int x, k=-1 ;
for (x=0 ;;x++) {
if fc(x) k++ ;
if (k==n) return x ;
}
}
```

Cette fonction donne bien le $n^{ième}$ élément de E.

De façon équivalente
Pour chaque x si fc(x) alors afficher x ;

Réciproquement supposons que f énumère les éléments de E. Peut-on calculer fc ?

Remarquons que

```
int fc (int x) {
int n ;
for (n=0 ;;n++) if f(n)==x return 1 ;
}
```

calcule la fonction suivante
X'(x)=1 si x∈ E et est non définie sinon.

Cette fonction est appelée fonction semi-caractéristique de E.

On soupçonne qu'il existe des ensembles récursivement énumérables et non décidables mais pour en construire un il faut faire intervenir un autre concept : le temps.

2. **Rôle du temps.**

h(p,x) =1 si p(x) est défini et 0 sinon. On a vu que h n'est pas calculable. Surchargeons l'opérateur h : h(p,x,t)=1 si p(x) est fini après t étapes de calcul 0 sinon. Le calcul du temps dépend du modèle de calcul mais on ne rentre pas ici dans le détail de cette question. Nous exigeons seulement deux axiomes :

(i) la fonction (p,x,t) -> h(p,x,t) est calculable (contrairement à la version avec deux variables).
(ii) h(p,x)=1 si et seulement s'il existe t avec h(p,x,t)=1.

Thérème. H={(p,x) tel que p(x) est défini} est ré. (et indécidable).

Preuve.
Pour chaque triplet (p,x,t) si h(p,x,t)=1 alors afficher (p,x)

Comme h est totale et calculable et que les triplets sont effectivement énumérables (voir chapitre 2), l'algorithme précédent permet d'énumérer les éléments de H. De plus H n'est pas décidable car sinon le problème de l'arrêt serait décidable.

3. **Un ensemble non ré.**

L'ensemble des parties de N n'est pas dénombrable. Or les procédures d'énumération sont en nombre dénombrable donc certains ensembles ne sont pas ré. Nous allons dans cette partie en donner un.

Théorème. Si E et son complémentaire sont tous deux ré alors E est décidable.

Preuve. Soient f la fonction d'énumération de E et g la fonction d'énumération du complémentaire de E.

Voici une procédure qui calcule la fonction caractéristique de E

```
int fc (int x) {
int n ;
(for n=0 ;; n++) {
if f(n)==1 return 1 ;
if  g(n)==1 return 0 ;
}
}
```

Conclusion : Le complémentaire de H (l'ensemble des couples (p,x) tels que p(x) n'est pas défini) n'est pas récursivement énumérable.
Preuve. Sinon d'après le théorème précédent H serait décidable.

4. **Ensembles semi décidables.**

On a vu à la fin de la section 1 que la fonction semi-caractéristique d'un ensemble ré. est calculable.
Appelons semi-décidabilité cette propriété qui est équivalente à la suivante :
E est semi-décidable ssi E=domaine(f) où f est une fonction calculable.

Théorème. E est semi-décidable ssi E est ré.

Preuve. Dans la section 1 on a vu la procédure qui calcule la fonction semi-caractéristique d'un ensemble ré. La réciproque est plus délicate :
Par exemple

```
for (x=0 ,,x++){
y=f(x) ;
afficher x ;
}
```
échoue .

On peut contourner en utilisant une nouvelle fois le temps :
Pour chaque couple (x,t) si h(f,x,t) alors afficher x ;

Applications : Soient f et g, deux fonctions calculables. Soit E={x| f(x) est défini et f(x)=g(x)}. Montrez que E est récursivement énumérable. E est-il calculable ?

Soient g une fonction calculable et totale et A={x| g(x)<x}. A est-il calculable ?
Même question si g est une fonction calculable mais pas forcément totale. Dans ce cas A est récursivement énumérable ? Donner un algorithme qui affiche les éléments de A.
Soit H le complémentaire de l'ensemble A. Caractériser les éléments de H. Que peut-on déduire si H est récursivement énumérable. H est-il récursivement énumérable ?

Donner un algorithme qui permet d'afficher l'ensemble des couples d'entiers. En déduire un algorithme qui permet d'afficher l'intersection de deux ensembles récursivement énumérables.
Que pensez-vous de la différence de deux ensembles récursivement énumérables ?