

## Introduction 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 qu'est-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  $\lambda$ -calcul, Kurt Gödel qui a montré qu'il existe des propositions vraies indémontrables en arithmétique, Alan Turing pour ses travaux sur le déchiffrement ...).

Thèse de Church. La réponse à la question est indépendante de la technologie. Tous les modèles de calcul 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 (compiler) dans un autre.

$\lambda$ -calcul  $\approx$  machine de Turing  
 $\approx$  langage de programmation raisonnable  
 $\approx$  langage C

Exemple. Peut-on déterminer à la compilation si un programme va boucler ou non ? La réponse est non dès que l'on a un langage de programme raisonnable (par exemple le langage C)

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 ? Quels problèmes peut-on (ou pourra-t-on) résoudre en temps raisonnable ?

**La théorie de la complexité** essaie de traiter cette question.

Prenons le cas de la partition avec 100 (resp.  $n$ ) objets, il y a  $2^{100}$  (resp.  $2^n$ ) partitions possibles. Avec un ordinateur qui fonctionne à 2 GHz depuis le début de la création de l'Univers on n'aurait pas exécuté plus de  $2^{90}$  instructions. Un algorithme en  $2^n$  n'est pas exécutable en pratique et donc 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. 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 avec 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  $\theta(2^n)$  on ajoute simplement 3 à la taille de la donnée.

## 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 est calculable  $\Rightarrow$  P calculable (si 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 f est calculable en temps polynomial, si le nombre d'appels à procQ est polynomial et tout ce qui est extérieur à cet appel est 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.

De même sous les mêmes hypothèses

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

P difficile  $\Rightarrow$  Q difficile (un algorithme polynomial pour Q entraînerait un algorithme polynomial pour P).

## 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  $\mathbb{N}$  dans  $\mathbb{N}$ .

#### Entiers relatifs

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

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

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

C'est bien une bijection (de plus sa programmation et la programmation de la fonction inverse est facile).  $\mathbb{N}$  et  $\mathbb{Z}$  ont la même cardinalité. On remarque que le programme  $x=0$  ; while (1) { écrire(C(x) ; x++; } 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	
$z=c(x,y)$	0	1	2	3	4	5	6	7	8	

Soit  $c(x, y) = (x+y)(x+y+1)/2 + y$

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).

## Généralisation au codage de triplets (ou de nuplets)

On peut donc coder les triplets par  $h(x,y,z) = c(c(x,y),z)$  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  $\sigma$  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 pouvant 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, on ne garde 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  $n^{\text{ième}}$  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  $\mathbb{N}$ ). Ces ensembles ont aussi la propriétés d'être énumérables par un algorithme. En fait on verra que de nombreux ensembles dénombrables (en bijection avec  $\mathbb{N}$ ) 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. 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  $F$  des fonctions totales de  $\mathbb{N}$  dans  $\mathbb{N}$  n'est pas dénombrable.

Preuve. Supposons que l'on a une bijection de  $\mathbb{N}$  dans  $F$ , en posant pour simplifier  $f(i)=f_i$  on a donc  $F=\{f_0, f_1, f_2 \dots f_i \dots\}$ . Soit la fonction  $\gamma(n)=f_n(n)+1$ . Cette fonction est bien une fonction totale de  $\mathbb{N}$  dans  $\mathbb{N}$  et donc appartient à  $F$ . 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.

Donc  $f$  n'existe pas.

### 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  $f_p$  (pas forcément défini en tout point).

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  $p_0, p_1, p_2 \dots p_n \dots$  les procédures et définissons  $\gamma$  par  $\gamma(n)=p_n(n)+1$ .

Cette fonction  $\gamma$  est calculable car à partir de  $n$  on peut calculer  $p_n$  et l'exécuter sur la donnée  $n$ .  $\gamma$  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  $\gamma$  n'est pas une fonction totale.

### Conclusion :

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

### 3. Fonctions calculables et totales

$f$  est calculable et totale s'il existe une procédure  $p$  qui la calcule et donc cette procédure se termine sur toute donnée (car la fonction est totale).

On ne peut pas les énumérer. En effet refaisons le même raisonnement : soit

$T = \{q_0, q_1, q_2 \dots q_n \dots\}$  les fonctions calculables totales. La fonction  $\gamma(n)=q_n(n)+1$  est totale. Avec les mêmes arguments que précédemment il ne peut pas y avoir égalité entre  $\gamma$  et  $q_i$ . Donc  $\gamma$  ne peut pas être calculable.

Conclusion : Soit  $T$  l'ensemble des procédures qui calculent des fonctions totales.  $T$  est dénombrable. Il existe donc des bijections de  $T$  dans  $\mathbb{N}$  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

Soit l'ensemble des procédures  $p_0, p_1, p_2 \dots p_n \dots$  et 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  $\gamma$ ) sont totales.

Si  $h$  est calculable alors il est facile de voir que  $\gamma$  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.

## 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 est indécidable.

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.

Si  $\delta$  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 :

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  $\text{proc}P$  qui calcule  $P$  alors on peut écrire

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

```
    else return  $p_1(x)$  ;  
}
```

Que vaut  $P(\text{gamma})$  ? 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.



## 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  $x \in E$  et 0 sinon.

$E$  est dit décidable si sa fonction caractéristique est calculable.

$E$  est récursivement énumérable si  $E=\{f(0), f(1), f(2), \dots\}$  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  $n^{\text{ième}}$  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  $x \in E$  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 les détails de cette question. Nous exigeons seulement deux axiomes :

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

Théorème.  $H = \{(p,x) \text{ tel que } p(x) \text{ est défini}\}$  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 énumérable (voir chapitre 2), l'algorithme précédent permet d'énumérer les éléments de  $H$ . De plus  $H$  n'est pas décidable car sinon le temps de l'arrêt serait décidable.

## 3. Un ensemble non ré.

L'ensemble des parties de  $\mathbb{N}$  n'est pas dénombrable. Or les procédures d'énumération sont en nombre dénombrables 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  $E = \text{domaine}(f)$  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  $x$  ;