Séance 5 : Fonctions récursives et machine de Turing

Objet : Le cours sera consacré aux fondements de l'informatique. Après avoir défini formellement la notion d'algorithme, on verra les deux modèles de calcul équvalents : fonctions récursives et machines de Turing. On aborde enfin quelques ques problèmes qui ne peuvent pas être résolus par un algorithme

---

Plan

- Fonctions primitives récursives
- Fonctions récursives partielles et récursives
- Machines de Turing
- Equivalence de deux modèles de calcul

V.1- Fonctions primitves récursives

On commence par étudier quelques exemples de problèmes fondamentaux de l'informatique et ensuite définir la sous-famille des fonctions primitives récursives qui sont des fonctions totales.

V.1.1. Introduction par les exemples

Dans la pratique des langages de programmation, on a souvent une impression que pour chaque
problème on peut trouver un algorithme de solution. Ce n'est pas le cas pour des nombreux problèmes naturels et intéressants il n'existe pas d'algorithme. Nous commencerons par quelques exemples illustratifs des problèmes de décision. Pour le moment nous ne donnons aucune preuve.

V.1.1.1- Problème de terminaison du programme "3x + 1"

Est-ce que le programme ci-dessous s'arrête sur un x donné ?

```
while x > 1 do   {
  if (x mod 2 = 0) then x := x/2;
else x := 3x + 1;
```

Réponse. Pour ce problème,
On peut utiliser la procédure suivante :

- à partir de x donné exécuter ce programme, et
- s'il s'arrête retourner "OUI".
- Par contre, si pour un certain x le programme "3x+1" boucle, cette procédure ne pourra jamais donner la réponse "NON".

Une telle procédure s'appelle un *semi-algorithme*.

Il est inconnu s'il existe un algorithme de décision pour ce problème, qui pour chaque x donné renvoie une réponse correcte "OUI" ou "NON". Une conjecture dit que ce programme s'arrête toujours. Si cette conjecture est vraie l'algorithme de décision serait tout simplement de retourner tout de suite "OUI" pour chaque x.

V.1.1.2- Problème de terminaison de programme :
C'est une généralisation du problème précédent.

Etant donné un texte d'une fonction (avec un argument entier) programmé en Java et un $x \in N$, est-ce que cette fonction s'arrte pour l'argument x ?

Pour ce problème il existe un semi-algorithme suivant :

Pour chaque x donné, exécuter f(x)
Si f(x) se termine alors écrire « OUI » sinon écrire « NON ».

Cela n'est pas un algorithme car pour certain x la procédure ne pourra jamais donner la réponse « NON » ..

V.1..1.3- Problème de correction de programme :

Etant donné un texte d'une fonction (avec un argument entier) programmé en C/Pascal une fonction. Est-ce que cette fonction calcule, par exemple la factorielle ?

Pour ce problème il n'existe ni un algorithme, ni même un semi-algorithme.
Donc le problème : «si un programme donné satisfait une spécification n'admet pas d'algorithme de décision .»

V.1..1.4- Problème de logique 1.
Est-ce qu'une formule propositionnelle F donnée (telle que $F = p \rightarrow g \vee p \vee \neg p$*) est valide ?*

Réponse.
Il existe des algorithmes pour ce problme, par exemple :

– Algorithme 1 : Construire la table de vérité. Si toutes les lignes contiennent seulement '1', répondre "OUI". Si dans une ligne il y a '0' répondre "NON".

V.1..1.5- Problème de logique 2
Est-ce qu'une formule du premier ordre données (telle que par exemple $\forall x P(x) \rightarrow \exists y P(y)$*) est valide ?*

Réponse. Il n'existe pas d'algorithme général (c-à-d il n'y a pas d'algorithme qui donne la bonne
réponse pour une formule du premier ordre quelconque). La procédure vue en cours de Logique (à
savoir, skolémiser la négation de la formule, la mettre en forme clausale et appliquer la résolution)
peut ne pas terminer.

V.1..1.6- Terminologies.

Dans ce cours, nous allons considérer des problèmes de décision qui peuvent être formulés comme suit.

Etant donné un ensemble U de toutes les données possibles (un univers) et un ensemble B de toutes les données dites "bien" (B $\subseteq$ U), pour chaque élément x $\subseteq$ U, répondre "OUI" si x $\in$B et "NON" si $\notin$ B. Par exemple, pour le problème de logique 2

U = {toutes les formules propositionnelles}
B ={toutes les formules valides} .

Par abus de notation nous écrirons ce problème comme "x $\in$ B ?", ou comme "B(x) ?"

Voici une définition informelle de problèmes décidables, semi-décidables, indécidables.

Définition V1.1.6.1

Pour un problème P = (U,B) donné,
– Le problème P est décidable , Il existe un algorithme pour P (c'est-à-dire une procédure qui s'arrête et répond "OUI" si l'entrée x $\in$B et répond "NON" si l'entrée x 6$\notin$ B).
– Le problème P est indécidable , Il n'existe pas d'algorithme pour P.
– Le problème P est semi-décidable , Il existe un semi-algorithme pour P (c'est-à-dire une procédure telle que si l'entrée x$\in$B elle r´epond "OUI" ; si l'entr´ee x $\notin$B dit "NON" ou ne donne pas de réponse).
Il est clair qu'un problème décidable est aussi semi-décidable.

Remarque.

- Pour montrer qu'un problème est décidable, il suffit de trouver un algorithme (un seul suffit et ceci peut se faire sans utiliser la théorie de la calculabilité).
- Par contre, pour montrer qu'un problème est indécidable, il faut considérer tous les algorithmes possibles et montrer qu'aucun d'eux ne résout le problème, ce qui est plus difficile et impossible sans théorie et sans notion rigoureuse d'algorithme.

Nous représentons une brève comparaison entre ces deux disciplines sous la forme d'un tableau :

	Calculabilité	Complexité
Question :	Existe-t-il un algorithme ?	Existe-t-il un algorithme rapide ?
Hypothèse :	Ressources non-bornées	Ressources bornées
Technique 1 :	Diagonalisation	Diagonalisation
Technique 2 :	Réduction calculable	Réduction polynômiale

V.1.2- Fonctions primitives récursives

V.1.2.1 Fonction caractéristique

Nous allons restreindre notre attention aux algorithmes pour calculer les fonctions de la forme f : $N^k \rightarrow N$, et nous voulons aboutir à une définition formelle des fonctions calculables f : $N^k \rightarrow N$.

Pour voir le lien entre la décidabilité et la calculabilité, considérons un probl`eme de d´ecision

(U,B) où U = N^k. Le probl`eme est donc le suivant : pour x∈ U d´ecider si x∈ B. Afin de décider si une entrée x ∈B, nous allons utiliser la fonction caractéristique de l'ensemble B (d'entrées "bien")
$\chi_B(x) : U \rightarrow N$, qui est définie comme suit.

Définition V.1.2.1- Fonction caractéristique

$$\chi_B(x) = \begin{cases} 1 & si\ x \in B, \\ 0 & sinon. \end{cases}$$

Définition V.1.2.2- Fonction calculable
Le problème (U,B) est décidable si et seulement si χB est calculable.

V.1.2.2 Fonctions récursives primitives
nous définissons dans cette partie la première classe des fonctions calculables. On obtiendra une classe assez large, appelée fonctions récursives primitives (RP), contenant presque tous les algorithmes que nous connaissons, mais, néeanmoins insuffisante.

Définition V.1.2.2.1- Fonction récursives primitives
La classe des fonction récursives primitives (RP) est la classe de fonctions f : Nk → N définie inductivement en utilisant 3 fonctions de base et 2 constructeurs suivants.
– Fonctions de base
- *– Zéro O : N^0 → N telle que O() = 0.*
- *– Successeur σ : N → N telle que σ(n) = n + 1.*
- *– Projecteur πki : N^k → N telle que πki (n1, n2, . . . , nk) = ni.*

– Constructeurs
- *– Composition Comp(f, g1, . . . , gk) = h t.q. h(x) = f(g1(x, . . . , gk(x)).*
- *– Rec(f, g) = u t.q.*
  - *u(m,0)= f(m)*
  - *u(m, n+1) = g(m, n, u(m,n))*

La classe RP est la plus petite classe qui contient les fonctions de base O, σ, πki et est fermée par les constructeurs Comp et Rec.

Exemple 1. ***Plus(x, y) = x + y***
Plus(x, 0) = x
Plus(x, n + 1) = Plus(x, n) + 1
Il suffit maintenant de récrire les équations ci-dessus sous forme Rec(f, g). Nous avons
Plus(x, 0) = π11(x) = f(x)
Plus(x, n + 1) = σ(π33(x, n, Plus(x, n))) = g(x, n, Plus(x, n))

Donc,
Plus = Rec(π11,Comp(σ, π33)).

Exemple 2. ***zero : (avec un argument).***
zero(0) = 0 = O() = f()
zero(n + 1) = zero(n) = π22(n, zero(n)) = g(n, zero(n))
donc ***zero = Rec(O, π22)***

V.1.2.2- Quelques propriétés de fermeture de la classe RP

Décrire les fonctions RP par des termes ne contenant que des fonctions de base et constructeurs Comp et Rec est une tâche fastidieuse. Nous établirons plusieurs propriétés de fermeture de la classe RP qui nous permettront d'établir la récursivité partielle plus rapidement.

Proposition V.1.2.2.1

Si g(x, i) est RP, alors f(x, n) = $\sum_{0\le i\le n}$ *g(x, i) et h(x, n) =* $\prod_{0\le i\le n}$ *g(x, i) sont aussi RP. Autrement dit, la classe de fonctions RP est fermée par rapport à* $\sum$ *et* $\prod$.

Preuve. Pour la somme $\sum$ c'est un exercice. Pour le produit $\prod$, on peut écrire la fonction h comme suit.

$h(x, 0) = g(x, 0)$

$h(x, n + 1) = h(x, n).g(x, n + 1)$

Puisque g et la multiplication . sont RP, la construction ci-dessus montre que h est aussi RP.

Exemple. Considérons $f(n) = \sum_{0\le i\le n} i$!

Afin de montrer que cette fonction est RP, notons d'abord que i! est RP puisqu'on peut l'exprimer comme

$0! = 1$

$(i + 1)! = i!\ (i + 1)$

Ensuite, f peut être écrite comme

$f(0) = 1$

$f(n + 1) = f(n) + (n + 1)!$

La fonction f est donc RP.

Remarque : prorammation les fonction RP

Programmer les fonctions de base en un langage de programmation comme Pascal est trivial.

En ce qui concerne la fonction Comp, si l'on sait programmer h(x1, . . . , xk) et g1(x), . . . , gk(x), on peut alors programmer h(x) = f(g1(x), . . . , gk(x)).

Pour la fonction Rec, un programme pour la calculer est le suivant :

```
fonction u(x, y)
r ← f(x)
for i = 0 to y − 1 r ← g(x, i, r)
return r
```

On peut donc voir qu'une fonction RP est programmable en utilisant seulement des boucles for imbriqués et les appels des fonctions non-récursifs. Or, le langage RP est équivalent à une partie du langage Pascal seulement avec la boucle for et sans appels récursifs.

V.1.2.3- Quelques exemples de fermeture de la classe RP

Exemple 3. ***Prédécesseur***

La fonction *prédécesseur* p définie par p(n) = max(0, n-1) est primitive récursive. Elle peut en effet en effet être définie de la manière suivante.

p(0) = 0
p(n+1) = n

p(0) = 0 = O()
p(n+1) = n = σ(π22(n, p(n)) = g(n, p(n))
donc ***p = Rec(O, σ(π22))***

Exemple 4. ***différence***
La fonction sub définie par sub(n, m) = n-m si n ≥ m et 0 sinon est aussi primitive récursive. Elle peut en effet en effet être définie de la manière suivante.

sub(n, 0) = n
sub(n, m+1) = p(sub(n, m))

sub(n,0) = n = π11(n) = f(n)

sub(n, m+1) = p(sub(n, m)) =comp(

Comme la récurrence porte sur le second argument, la fonction sub est égale à sub'(p_2, p_1) où la fonction sub' est égale à Rec(p_1, p(p_2)).

Exemple 5. ***Produit***

La fonction prod définie par prod(n, m) = n*m est primitive récursive. Elle peut en effet en effet être définie de la manière suivante.

- prod(0, m) = 0
- prod(n+1, m) = sum(prod(n, m), m)

La fonction somme est donc égale à Rec(0, sum(p_2, p_3)).

Exemple 6. ***Égalité***

La fonction eq0 définie par eq0(m) = 1 si m = 0 et eq0(m) = 0 sinon est primitive récursive. Elle est égale à Rec(1, 0). La fonction eq définie par eq(m, n) = 1 si m = n et eq(m, n) = 0 sinon est primitive récursive. Elle est égale eq0(sum(sub(p_1, p_2), sub(p_2, p_1))).

Exemple 7. ***Division et reste***

Les fonctions div et mod où div(n, m) et mod(n, m) sont respectivement le quotient et le reste de la division entière de n par m sont primitives récursives. La fonction div peut être définie de la manière suivante.

- div(0, m) = 0,
- div(n+1, m) = div(n, m) + eq(m*(1+div(n, m)), n+1)

La fonction mod peut être alors définie par mod(n, m) = n - m*div(n, m).

Exemple 8. ***Puissance, racine et logarithme***

La fonction pow où pow(n, m) = m^n est primitive récursive. Elle peut être définie de la manière suivante.

- pow(0, m) = 1,
- pow(n+1, m) = prod(pow(n, m), m)

La fonction root où root(n, m) est la racine m-ième de n, c'est-à-dire le plus grand entier k tel que $k^m \leq n$ est primitive récursive. Elle peut être définie de la manière suivante.

- root(0, m) = 0,
- root(n+1, m) = root(n, m) + eq(m, pow(1+root(n, m)), n+1)

La fonction log où log(n, m) est le logarithme en base m de n, c'est-à-dire le plus grand entier k tel que $m^k \leq n$ est primitive récursive. Elle peut être définie de la manière suivante.

- log(0, m) = 0,
- log(n+1, m) = log(n, m) + eq(pow(1+log(n, m), m), n+1)

Nombres premiers

On commence par définir la fonction ndiv(n) qui donne le nombre de diviseurs de l'entier n. Cette fonction est définie par ndiv(n) = pndiv(n, n) où la fonction pndiv qui donne le nombre de diviseurs de n inférieurs à p est définie de la manière suivante.

- pndiv(0, n) = 0,
- pndiv(p+1, n) = pndiv(p, n) + eq0(mod(n, p+1)).

La fonction premier est alors définie par premier(n) = eq(ndiv(n), 2).

Valuation

La fonction val où val(n, m) est le plus grand entier k tel que m^k divise n est primitive récursive.

V.1.3- Les prédicats récursifs primitifs

Définition V.1.3.1- Rappel de Prédicat
Un prédicat sur N^k d´ecrit une propri´et´e d'un k-uplet (x1, . . . , xk). Il peut être d´efini par une fonction $N^k \rightarrow v\{vrai, faux\}$ ou bien par un sous-ensemble de N^k.

Exemple 1.
Le prédicat *Pair* sur N. Nous avons *Pair*(5) = faux et *Pair*(6) = vrai. Le prédicat *Pair* correspond à l'ensemble {0, 2, 4, 6, 8 . . .}.

Exemple 2.
Le prédicat *PlusGrand* sur N^2 est défini comme suit.

PlusGrand(x, y) = vrai si x > y,
faux sinon.

Il correspond à un ensemble {(1, 0), (2, 0), . . . , (2, 1), (3, 1), . . .}.
Dans le cadre défini dans l'introduction un prédicat P sur $\mathbb{N}^k$ correspond au problème (U,B) avec U = $\mathbb{N}^k$ et B = {x|P(x)}.

Définition V.1.3.2- Rappel de Fonction caractéristique
Soit P un prédicat sur $\mathbb{N}^k$. Sa fonction caractéristique est $\chi P : \mathbb{N}^k \rightarrow N$ telle que
$\chi P(x) =$ *1 si P(x),*
0 si ¬P(x).

Exemple 3
La fonction caractéristique du prédicat *Pair* est donc
$\chi Pair(2n) = 1$,
$\chi Pair(2n + 1) = 0$.

Et celle du prédicat *PlusGrand*
$\chi PlusGrand(x, y) =$ 1 si x > y,
0 sinon.

Définition V.1.3.3- Prédicat RP (récursif primitif)
Un prédicat P sur $\mathbb{N}^k$ est RP ssi χP est RP.

Exemple 4.
La fonction caractéristique d'un prédicat P tel que $P(x) \Leftrightarrow x \neq 0$ est :

$\chi P(x) =$ 1 si x > 0,
0 si x = 0.

On peut voir que $\chi P(x) = sig(x)$ qui est RP et χP est donc RP.
Par conséquent, le prédicat P est RP.

Proposition V.1.3.1
Soient P et Q deux prédicats RP, alors $P \wedge Q$, $P \vee Q$, $\neg P$ et $P \rightarrow Q$ sont aussi RP.

Autrement dit, la classe de pr´edicats RP est ferm´ee par les op´erations bool´eennes.

Preuve.
Nous prouvons d'abord que $P \wedge Q$ est RP.

Comme $(P \wedge Q)(x) \equiv P(x) \wedge Q(\bar{x})$,

alors,
$\chi P \wedge Q(x) =$ 1 si $\chi P(x) = 1$ et $\chi Q(x) = 1$,

0 sinon.
On peut en déduire que $\chi P \wedge Q(x) = \chi P(x).\chi Q(x)$.

Or χP et χQ sont RP par hypothèse et, de plus, on sait que la multiplication est RP. Par conséquent, $\chi P \wedge Q$ est RP et $P \wedge Q$ l'est aussi.

Nous allons maintenant prouver que $\neg P$ est RP.

$\chi_{\neg P}(x) =$ 1 si $\chi_P(x) = 0$,
0 si $\chi_P(\bar{x}) = 1$.
Il est facile de voir que $\chi_{\neg P}(x) = 1 - \chi_P(x)$.
Alors, $\chi_{\neg P}$ est RP et $\neg P$ est donc RP.
Pour "$\vee$" et "$\rightarrow$", on peut les exprimer en utilisant "$\wedge$" et "$\neg$" comme suit :

$(P \vee Q) \equiv (\neg(\neg P \wedge \neg Q))$ et

$(P \rightarrow Q) \equiv (\neg(P \wedge \neg Q))$.

Par la suite, nous présentons les propriétés de la RP.

Proposition V.1.3.2 : Quantification bornée
Soit P(x, y) un prédicat RP. Alors, $\forall x \leq n\ P(x, y)$ et $\exists y \leq n\ P(x, y)$ sont RP.

Il est important de noter que cette propriété n'est pas vraie pour une quantification arbitraire, par exemple $\forall x P(x, y)$ ou bien $\exists y P(x, y)$.

Preuve
Pour $\forall\leq$, notons $Q(x, n) = \forall x \leq n : P(x, y)$. Donc,

$\chi_Q(x, n) =$ 1 si $\forall y \leq n : \chi_P(x, y) = 1$,

0 sinon.

$= \prod_{y=0_{\leq n}} \chi_P(x, y)$

On sait que χ_P est RP, par hypothèse. En plus, selon la Proposition 1, le produit $\prod_{y=0,..n}$ préserve la récursivité primitive. Ainsi, RP est fermé par $\forall\leq$.

Pour $\exists\leq$, il suffit d'utiliser la propriété que $\exists x \leq n\ P(x, y)$ peut être exprimé comme $\exists x \leq n$

$P(x, y) \equiv \neg(\forall x \leq n : \neg P(x, y))$

Proposition V.1.3.3 :
Si les fonctions gi et les pr´edicats Pi sont RP (pour $i \in \{1, \ldots, n\}$, alors la fonction

f(x) = *g1(x) si P1(x),*
...
gn(x) si Pn(x).
est aussi RP.

Preuve :
Pour prouver la proposition, notons que la fonction f peut être écrite comme

f = g1.χP1 + . . . + gn.χPn.

Alors, f est RP puisqu'elle est obtenue à partir des fonctions RP en utilisant l'addition et la multiplication.

Définition V.1.3.4 : Opérateur de minimisation bornée
Soit P(n,m) un prédicat. Alors,

$f(n,m) = \mu^{i \leq m} P(n, i)$ = *le plus petit i t.q. P(n,m) s'il existe*
= *0 si un tel i n'existe pas*

On dit que f est obtenue de P par la minimisation bornée.

Proposition V.1.3.4 : Fermeture de RP par minimisation bornée
Si P est RP, et f est obtenue de P par la minimisation bornée, alors f est aussi RP

Preuve.
Soit $f(n,m) = \mu^{i \leq m} P(n, i)$.
On peut définir la fonction f par cas :

f(n,m) = 0 si P(n, 0)
f1(n,m) sinon.

La fonction f1, qui correspond au cas non-trivial de la minimisation quand P(n, 0) est faux, peut être définie par récurrence primitive :
f1(n, 0) = 0
f1(n,m + 1) = f1(n,m) si f1(n,m) > 0
m + 1 si f1(n,m) > 0 ∧ P(n,m + 1)

0 sinon

La première ligne du cas récursif correspond au cas où le plus petit i satisfaisant la propriété est ≤ m (et donc nous l'avons déjà trouvé), la deuxième correspond au cas où le plus petit i est m+1 (et donc nous ne l'avions pas encore trouvé aux étapes précédentes, mais P(¯n,m+1) est vrai), la dernière est pour le cas où nous ne trouvons pas le bon i ni avant, ni à l'étape m + 1. Toutes les fonctions et prédicats qui interviennent dans la définition de f1 sont RP, et, par conséquent, f1 l'est aussi. Donc f est aussi RP.

V.1.4- Limite des fonctions primitives récursives

Une première limitation de la récursion primitive intervient dans les algorithmes susceptibles de ne pas se terminer. Tel est le cas de la quantification non bornée ou de la minimisation non bornée, vues précédemment.

Mais il ne suffit qu'une fonction soit définie récursivement, et par un procédé se terminant pour toute valeur des données, pour que la fonction soit récursive primitive. L'ensemble des fonctions récursives primitives n'est en effet qu'une partie de l'ensemble des fonctions récursives. Ainsi, la fonction d'Ackermann A de N^3 dans N définie par

- A(k, m, n) = n+1 si k = 0,
- A(k, m, n) = m si k = 1 et n = 0,
- A(k, m, n) = 0 si k = 2 et n = 0,

- A(k, m, n) = 1 si k ≠ 1, 2 et n = 0,
- A(k, m, n) = A(k-1, m, A(k, m, n-1)) sinon.

Proposition. *Pour tous m et n,*

- *A(1, m, n) = m+n,*
- *A(2, m, n) = m*n,*
- *A(3, m, n) = m^n,*
- *A(4, m, n) = $m^{m^{\cdots^n}}$.*

Une version simplifiée de la fonction de N^2 dans N peut être obtenue en prenant m = 2 dans la définition ci-dessus. On prend aussi souvent comme définition la définition suivante qui est légèrement différente.

- A(0, n) = n+1,
- A(k+1, 0) = A(k, 1),
- A(k+1, n+1) = A(k, A(k+1, n)) sinon.

On dit qu'une fonction g de N dans N majore une fonction f de N^k dans N si $f(n_1, \ldots, n_k) \leq g(\max(n_1, \ldots, n_k))$.

Proposition. *Pour toute fonction primitive récursive f de N^k dans N, il existe un entier k tel que f est majorée par la fonction g(n) = A(k, n).*

Corollaire. *La fonction d'Ackermann n'est pas primitive récursive.*

V.2- Fonctions récursives partielles et récursives

Les fonctions récursives constituent une autre façon de définir la notion de *calculabilité*. Elles sont définies à partir de quelques fonctions de base, de la composition, d'un schéma de récurrence et d'un schéma de minimisation.

Les fonctions récursives sont des fonctions, éventuellement partielles, qui calculent des n-uplets d'entiers naturels. Chaque fonction récursive est une fonction d'un sous-ensemble de N^k dans N^r pour des entiers k et r.

V.2.1- Fonctions partielles

Nous avons vu que la classe de fonctions RP est en fait une sous-classe de fonctions calculables. La question qui se pose ensuite est comment construire la vraie classe de fonctions calculables, même si l'on élargit la classe de fonctions de base et les opérations.

Définition V2.1.1- Fonction partielle
Une fonction partielle $f : N^k \cdots \rightarrow N$ est une fonction d'un sous-ensemble de N^k vers N. Le domaine de f est Dom(f) = {x | f(x) est définie}.
- *Si en x la fonction f est définie, on écrit f(x) ↓ (on lit : f converge sur x),*
- *sinon on écrit f(x) ↑ ou f(x) = ⊥ (f diverge).*
- *Si Dom(f) = N^k on dit que f est totale.*

Par convention, quand on applique des opérations arithmétiques (ou d'autres fonctions) à une
valeur indéfinie ⊥, le r´esultat diverge également.

Exemple 1.

f(x) = x/2 si x est pair,
↑ sinon

Alors,
f(5) + 1 ↑,
f(6) + 1 = 3 + 1 = 4, et
0.f(7) + 2 ↑.

En général, si f est une fonction partielle, nous avons
0. f(x) = 0 si f(x) ↓
↑ si f(x) ↑

V.2.2- Fonctions récursives partielles

Pour définir les fonctions récursives partielles, on introduit un schéma de minimisation, appelé minimisation non-bornée.

Définition V.2.2.1- Minimisation (non bornée)

Soit k est un entier et soit f une fonction de N^{k+1} dans N éventuellement partielle. On définit alors la fonction g = Min(f) de N^k dans N de la manière suivante :
- Pour m dans N^k, g(m) est l'entier n, s'il existe, tel que d'une part les valeurs f(m,0), …, f(m, n-1) sont égales à 0 et d'autre part f(m, n) est défini et différent de 0.
- Si un tel entier n n'existe pas, g(m) n'est pas défini. Même si la fonction f est totale, la fonction g = Min(f) peut être partielle.

Exemples de minimisation

1/ $\mu y.((y + 1)^2 > x) = \lfloor \sqrt{x} \rfloor$
2/ $\mu y.(y^2 = x) =$ $\sqrt{x}$ si x est un carré parfait,
↑ sinon.
3/ $\mu k.(2k = x) =$ x/2 si x est pair,
↑ sinon.

Proposition V.2.2.1 : Si f est calculable totale, alors la minimisation est aussi calculable.
Preuve. Afin de prouver la proposition, nous montrons le programme suivant qui calcule g. Notons que pour programmer la minimisation non-born´ee, il faut utiliser des boucles while.

```
fonction g(x)
y ← 0
while (f(x, y) 6= 1)
 { y ← y + 1}
return y
```

Définition V2.2.2- fonctions récursives (partielles)
La famille des fonctions récursives (partielles) *est la plus petite famille de fonctions qui vérifie les conditions suivantes.*

1. *Les fonctions primitives récursives sont récursives (partielles).*
2. *La famille des fonctions récursives (partielles) est close pour les schémas de composition, de récurrence et de minimisation.*

Proposition V.2.2.2 : Thèse de Church
La classe de fonctions récursives (partielles) est égale à la classe de fonctions calculables par un algorithme quelconque.

Remarque :
Notons que ceci est une thèse (et pas un théorème) puisque la classe de fonctions récursives partielles est formellement déefinie tandis que "la classe de fonctions calculables" est une notion intuitive et informelle.
La thèse de Church contient deux parts :
1. Chaque fonction récursive partielle est calculable.
2. Chaque fonction calculable est récursive partielle.

V.3- Machines de Turing

Nous allons maintenant étudier un autre modèle de calcul qui est la machine de Turing. Ce modèle est très différent du modèle fonctionnel des fonctions récursives partielle. Il ressemble plus aux modèles de calcul de type "machine", tels que des automates ou des machines à pile (et aux ordinateurs réels).
La raison pour laquelle nous concentrons sur les machines de Turing est qu'elles sont des machines abstraites très simples mais capables de réaliser tous les algorithmes.
Pour mieux placer les machines de Turing dans le contexte nous pouvons considérer d'abord des modèles plus simples :
- Un automate fini, n'ayant pas de "mémoire externe", a une capacité de calcul assez limitée. - Une machine à pile est un automate avec une pile, qui représente une mémoire externe non-bornée, mais avec l'accès très restreint (seulement par le sommet).

Une machine de Turing peut être vue comme un automate avec
- un ruban de longueur infinie (qui est une mémoire avec des règles d'accès assez libérales). Plus précisément,
- chaque case du ruban contient un symbole (on suppose que seulement un nombre fini de cases contiennent un symbole non-nul),
- la machine possède une tête qui peut se déplacer sur le ruban, lire le symbole à sa position, et réécrire ce symbole. La définition formelle est la suivante.

Définition V.3.1 : Machine de Turing
Une machine de Turing (une MT) est un quadruplet MT = (Q, Σ, q0, δ) tel que
– Q : un ensemble fini appelé l'ensemble d'états,
– Σ : un alphabet fini appelé l'alphabet du ruban (on suppose que cet alphabet contient 0 et 1),
– q0 : un élément de Q appelé l'état initial,
– δ : le programme qui est un ensemble d'instructions.
Les instructions ont la forme suivante :
(état, symbole lu) → (état, symbole écrit, d´élacement)
où
- *les deux états appartiennent à Q,*

- *le deux symboles à ∑, et*
- *le déplacement à{G,D, −}.*

Il y a donc trois types d'instructions :
– (q, a) → (p, b,G) : a en état q passer à l'état p, écrire b, déplacer la tête à gauche,
– (q, a) → (p, b,D) : déplacer à droite),
– (q, a) → (p, b,−) : aucun déplacement

Définition V.3.2- Configurations et calculs d'une machine de Turing
1/ ***Une configuration*** *d'une MT est (wG, q,wD)*
où
- wG est le mot à gauche de la tête de lecture jusqu'au premier symbole non-nul,
- q est l'état actuel,
- wD est le mot à droite de la tête de lecture y compris la case où la tête se trouve, et jusqu'au dernier symbole non-nul.
2/ ***Un calcul*** *de la machine sur une entrée (x1, . . . , xk) ∈ N^k est une séquence de*

configurations c1, c2, . . . , cn telle que
i) Chaque configuration c_{i+1} est obtenue à partir de la configuration ci en appliquant une instruction de la machine.
ii) c1 = ε, q0, $01^{x1}01^{x2}$ . . . 01^{xk} .
iii) Dans la configuration cn il n'y a pas d'instructions applicables.

Hypothèse : *Dorénavant, nous considérons seulement les machines de Turing déterministes.*

Exemple 1 : Machine de Turing
Supposons que la machine M a seulement 2 instructions :
I1 : (q0, 0) → (q1, 1,D)
I2 : (q1, 0) → (q1, 3,G)

et qu'initialement le ruban de la machine M ne contient que des 0 :
... . . | 0 | 0 | 0 | 0 | 0 | 0 | . . .
△

Alors, après avoir exécuté le programme, la machine est dans la configuration suivante :
;;. . . | 0 | 0 | 1 | 3 | 0 | 0 | . . .
△

q0

Le calcul de la machine est donc (ε, q0, ε) → (1, q1, ε) → (ε, q0, 3).

Définition V.3.3- Fonctions calculées par les machines de Turing
Le résultat de calcul de la machine M est le nombre de "1" sur le ruban quand la machine est dans la configuration cn. La fonction(partielle) $f_{k, M}$ de k arguments calculée par la machine M est donc définie comme suit :
- Si sur l'entrée (x1, . . . , xk) la machine M s'arrête dans une configuration cn alors
f(x1, . . . , xk) = nombre de "1" dans cn.
- *si elle ne s'arrête pas, alors f(x1, . . . , xk) ↑*

V.4- Equivalence entre les Fonctions Récursives et les Machines de Turing

Nous présentons maintenant les résultats importants qui montrent le lien entre les fonctions calculées par les machines de Turing et les fonctions récursives (partielles).

Théorème V.3.1-
Chaque fonction récursive (partielle) peut être calculée par une machine de Turing.

Théorème V.3.2-
Chaque fonction calculée par une machine de Turing est récursive (partielle).

Remarque : forme équivalente
Une fonction de N^k dans N est récursive si et seulement si elle est calculable par une machine de Turing.

D'où la forme équivalente de la thèse de Church

Thèse de Church-Turing.
La classe de fonctions calculables par machines de Turing est égale à la classe de fonctions calculables par un algorithme quelconque.

Pour les machines de Turing, il faut préciser comment sont codés les entiers. Ils peuvent être codés en binaire, en base 10 ou en base 1. Le choix du codage est en réalité sans importance puisqu'il est possible de passer d'un quelconque de ces codages à n'importe quel autre avec une machine de Turing.

Il est facile de de se convaincre qu'une fonction récursive est calculable par une machine de Turing. Les fonctions de base comme les fonctions constantes, les projections, les fonctions de duplication sont bien sûr calculables par machine de Turing. Ensuite avec des machines calculant des fonctions f, g, f_1, …, f_p, il n'est pas difficile de construire des machines de Turing pour les fonctions $g(f_1, \ldots, f_p)$, Rec(f, g) et Min(f).

Invesement, nous pouvons également montrer qu'une fonction calculable par une machine de Turing est récursive.

Exemple 2 : Si f et g1, . . . , gk sont calculables par une MT, alors Comp(f, g1, . . . , gk) est calculable par une MT.

Preuve.
Nous considérons uniquement le cas de deux fonctions scalaires. Soient f et g deux fonctions MT-calculables : $f = f_{1,R}$ (calculable par la machine R), $g = f_{1,S}$ (calculable parla machine S). Nous devons alors calculer f(g(x)).

Soient r0, r1, . . . , rn les états de R, et s0, s1, . . . , sm les états de S.
Nous supposons que ces deux ensembles sont disjoints.

Nous allons construire une machine de Turing M pour calculer f(g(x)). Les états de M sont

$$q0, q1, r0, r1, \ldots, rn, s0, s1, \ldots, sm$$

et ses instructions sont :

I1 : $q_0 0 \rightarrow s_0 0$ (lancer S),
δ_S (instructions de S),
I2 : $s_1 0 \rightarrow r_0 0$ (lancer R),
δ_R (instructions de R),
I3 : $r_1 0 \rightarrow q_1 0$ (stop).

Les étapes de calcul de M sont

$$(\varepsilon, q0, 01^x) \xrightarrow{I1} (\varepsilon, s0, 01^x) \xrightarrow{S} (\varepsilon, s1, 01^{g(x)}) \xrightarrow{I2} (\varepsilon, r0, 01^{g(x)}) \xrightarrow{R} (\varepsilon, r1, 01^{f(g(x))}) \xrightarrow{I3} (\varepsilon, q1, 01^{f(g(x))})$$

ce qui conclut la preuve.

V.5- Un peu de histoire

En théorie de la calculabilité (on dit aussi récursivité !), une fonction récursive est une fonction à un ou plusieurs arguments entiers, qui peut se calculer en tout point par une procédure mécanique, par un programme peut-on dire depuis l'avènement de l'informatique. Il est plus cohérent de définir les *fonctions partielles récursives,* qui sont aussi des fonctions partiellement récursives, avant les fonctions récursives. Formellement, une fonction récursive est alors une fonction partielle récursive définie en tout point. On trouve de plus en plus souvent le terme de *fonction calculable* pour désigner une fonction récursive, qui est le terme historique. Mais ce dernier est encore très utilisé. Il y a plusieurs définitions équivalentes de fonctions calculables, l'une telle étant que ce sont les fonctions calculées par une *machine de Turing* (ces fonctions sont a priori partielles) dont le calcul termine pour toute entrée.

Historiquement on a d'abord introduit dans les années 1920 la classe des *fonctions récursives primitives*, dont on s'est rapidement rendu compte qu'elle ne capturait pas toutes les *fonctions calculables* (en un sens encore intuitif). Le schéma de définition par récurrence utilisé pour définir ces fonctions est bien récursif au premier sens du terme : une fonction est définie en fonction d'elle-même.

Jacques Herbrand décrivit, dans une lettre adressée à Kurt Gödel en 1931, un modèle de calcul, des systèmes d'équations avec un mécanisme d'évaluation symbolique « par valeur », comme on dirait maintenant. Ce modèle fut précisé par Gödel lors d'exposés à Princeton en 1934. Les fonctions ainsi calculées furent appelées *fonctions récursives générales*. Les systèmes d'équations de Herbrand-Gödel utilisent de façon libre des appels récursifs de fonction au premier sens de ce terme. L'équivalence de cette définition avec celle des fonctions *λ-calculables*, et des fonctions calculables par *machine de Turing* fournit des arguments pour la *thèse de Church-Turing*, qui énonce que ces modèles capturent effectivement la notion intuitive de fonction calculable.

On utilise parfois plus spécifiquement le terme de fonctions récursives pour les *fonctions récursives au sens de Kleene*, ou fonction *μ-récursives*, l'une des définitions possibles de fonctions calculables, introduite par Kleene en 1936 qui laisse le calcul implicite. Cette définition généralise celle des fonctions récursives primitives.

Enfin les *théorèmes de point fixe de Kleene*, qui permettent entre autre de justifier l'utilisation des définitions récursives de fonctions (au sens premier de ce terme), sont également appelés *théorèmes de la récursion.*