DOMAINE : Non olympique
NIVEAU : Intermédiaire
CONTENU : Cours

AUTEUR : Vincent JUGÉ
STAGE : Montpellier 2014

Calculabilité

Ce cours a vocation à présenter aux élèves les concepts voisins de calculabilité et de complexité. Ces deux concepts sont reliés à la question suivante, très simple en apparence : étant donné un objet mathématique, est-il possible de le calculer ? Si oui, est-ce une opération difficile ?

1 Calculabilité

1.1 Qu'est-ce qu'un calcul ?

Et oui : qu'est-ce qu'un calcul ? Généralement, on s'imagine qu'un calcul, c'est ça :

$$(1+X)^n + (1-X)^n = \sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{2k} X^{2k}.$$

On se dit alors qu'on a « calculé » l'expression $(1+X)^n + (1-X)^n$. À moins que l'on n'ait calculé l'expression $\sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{2k} X^{2k}$, d'ailleurs.

On peut se rassurer en se disant qu'un calcul, ce n'est rien d'autre que mettre en évidence des relations entre différents objets mathématiques (ici, des polynômes) : une telle définition est peut-être insatisfaisante, mais elle a au moins l'avantage de faire jouer des rôles symétriques aux expressions situées de part et d'autre de l'égalité. Mais alors comment pourrait-on dire qu'une chose est ou n'est pas calculable ? D'ailleurs, étant donné deux objets égaux que l'on définit de manières différentes, peut-on nécessairement montrer qu'ils sont égaux ?

Cette question est en fait une question de logique : comment faire pour démontrer un théorème ? Généralement, on part de résultats de base, les **ax-**

iomes, qui paraissent évidents. Puis on en déduit des résultats moins basiques grâce à des **règles de déduction** que l'on considère comme tout aussi évidentes : par exemple, si A implique B et si B implique C, alors A implique C. On voit apparaître ici la notion de démonstration incrémentale : en exécutant une suite de petites modifications, très simples, locales, et en partant d'objets simples connus (les axiomes), on réussit à obtenir un théorème a priori non trivial. On peut donc dire qu'on a **calculé** une démonstration de notre théorème.

Un calcul, ce serait donc cela : une suite d'opérations que l'on aura considéré comme élémentaires, et qui nous permet de fabriquer un objet à partir d'un autre. Dans l'exemple précédent, on aura utilisé des opérations qui conservent le caractère « vrai » d'une proposition et on sera parti d'axiomes que l'on considérait comme vrais, donc on aboutira à un théorème que l'on pourra également considérer comme vrai.

Maintenant, on a certes une vision intuitive de ce que l'on devrait considérer comme un calcul, donc de ce que l'on devrait considérer comme calculable : est calculable un objet que l'on pourrait obtenir à partir d'objets de base, après application d'une suite d'opérations élémentaires. Mais, dans une telle définition, on n'a malheureusement pas défini ce qu'était un objet de base, ni même quelles opérations on pourrait considérer comme élémentaires. Il est donc temps d'étudier plus en détail un modèle de calcul très classique : la **machine de Turing**.

1.2 La machine de Turing déterministe

Au cours du siècle dernier, Alan Turing a élaboré un concept de machine à calculer universelle, dans le sens où elle pourrait calculer tout ce que l'on pourrait concevoir de calculer. Le revers de la médaille étant qu'une telle machine est peu évidente à fabriquer en pratique, et qu'elle serait affreusement lente. Voici le modèle qu'il a proposé. [1]

Une machine de Turing **déterministe** est composée de

- k **rubans** infinis (où k est un entier au moins égal à 2) ; chaque ruban contient une infinité de cases : une première (en position 0), puis une deuxième (en position 1), etc. de sorte que l'on peut identifier le ruban à $\mathbb{N}$;
- un **alphabet** fini, noté $\mathcal{A}$, qui regroupe les symboles qui peuvent être

1. Je remercie ici Gilles Dowek, dont j'ai adoré le cours sur la calculabilité, au point de m'inspirer très fortement ici du polycopié qu'il avait alors rédigé.

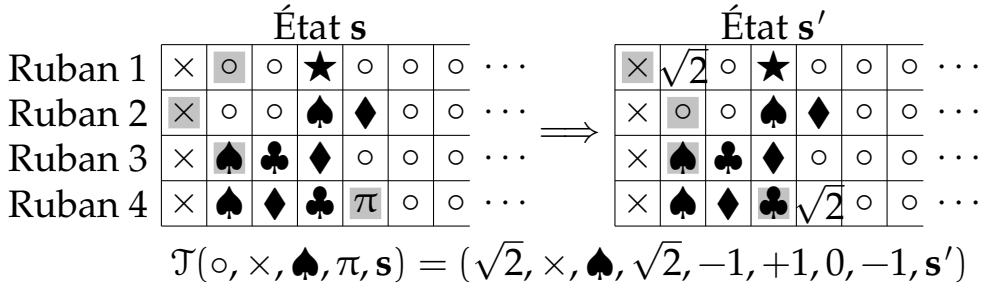
écrits sur chaque case de chaque ruban ; en particulier, cet alphabet devra contenir un symbole de **début de ruban** (noté **D**), qui permet de reconnaître que l'on est arrivé sur la première case du ruban, un symbole **blanc** (noté $\circ$), qui symbolise le fait qu'on n'a pas encore écrit quoi que ce soit sur la case en question ;

- k **têtes de lecture et d'écriture**, positionnées chacune sur leur ruban : elles peuvent écrire sur leur case, en lire la valeur, et se déplacer d'une case vers la droite ou vers la gauche ;
- un ensemble fini d'états, noté $\mathcal{S}$, dont un état **initial** (noté $\mathbf{s_i}$) et un état **final** (noté $\mathbf{s_f}$), qui signale la fin du calcul ;
- une **table de transition** finie, notée $\mathcal{T}$, qui est en fait une fonction $\mathcal{T} : \mathcal{A}^k \times \mathcal{S} \mapsto \mathcal{A}^k \times \{-1, 0, 1\}^k \times \mathcal{S}$ destinée à décrire l'évolution de la machine, d'une manière que l'on va expliquer ci-dessous.

Au départ, quand on lance la machine, on exige que seul un nombre fini de cases ne soit pas muni du symbole blanc : en effet, il est difficile de dire que l'on a déjà eu le temps de modifier un nombre infini de cases… Le fait qu'un nombre fini de cases ne soit pas blanc restera une constante durant toute l'exécution de la machine. D'autre part, on exige également que les têtes de lecture soit initialement positionnées en position 0, c'est-à-dire sur les cases $\times$: il faut bien décider d'une position canonique pour débuter le calcul.

Comment le calcul s'opère-t-il alors ? Et bien, à chaque étape, la machine est dans un état $\mathbf{s} \in \mathcal{S}$ et nos k têtes de lecture peuvent lire les valeurs $\lambda_1, \ldots, \lambda_k \in \mathcal{A}$ de leurs k cases respectives. On considère alors les k symboles $\mu_1, \ldots, \mu_k \in \mathcal{A}$, les k entiers $\varepsilon_1, \ldots, \varepsilon_k \in \{-1, 0, 1\}$ et l'état $\mathbf{s}'$ tels que $\mathcal{T}(\lambda_1, \ldots, \lambda_k, \mathbf{s}) = (\mu_1, \ldots, \mu_k, \varepsilon_1, \ldots, \varepsilon_k, \mathbf{s}')$. Et l'on demande à la tête située sur le i-ème ruban d'écrire le symbole μ_i (effaçant ainsi le symbole λ_i qu'elle vient de lire) puis de se déplacer de ε_i cases vers la droite (donc d'une case vers la gauche si $\varepsilon_i = -1$) ; la machine rentre alors dans l'état $\mathbf{s}'$.

Enfin, le calcul est fini si on atteint l'état final $\mathbf{s_f}$; notons que, si l'on n'atteint jamais cet état, la machine de Turing continuera son calcul *ad vitam æternam*. En outre, on voit là qu'il est en fait inutile de définir les valeurs de $\mathcal{T}(\lambda_1, \ldots, \lambda_k, \mathbf{s_f})$, puisqu'elles ne seront jamais utilisées. D'autre part, qu'aura-t-on calculé ? Et bien, à partir de l'état initial des k rubans, on a tout simplement calculé l'état final de ces k mêmes rubans.



Fonctionnement d'une machine de Turing

L'exemple ci-dessus montre l'état d'une machine de Turing au cours d'un calcul : ses têtes de lecture et d'écriture, représentées par des carrés gris, sont initialement en positions $(1,2)$, $(2,1)$, $(3,2)$ et $(4,5)$. La table de transition provoque alors une action de la part de la machine, de telle sorte qu'on se retrouve finalement dans la configuration de droite, avec des têtes de lecture et d'écriture en positions $(1,1)$, $(2,2)$, $(3,2)$ et $(4,4)$.

Pourquoi un tel modèle : qu'entendait Turing par « tout ce que l'on pourrait concevoir de calculer » ? Et bien, dans la vie, quand on veut faire un calcul, que peut-on faire ? On peut regarder les données que l'on a autour de nous, et utiliser un certain nombre de règles pour interagir avec ces données. Or, ces données apparaissent comme étant de type **fini** et **discret** : par exemple, sur un ordinateur, il s'agira des bits dans la mémoire.

Notons bien que cela n'a rien d'évident a priori, et que l'on pourrait vouloir travailler directement avec des objets infinis, par exemple des nombres réels, dont le développement décimal est a priori arbitraire. Cependant, pour des raisons pratiques, ce n'est pas un tel modèle qui a été choisi par Turing : remplacer un chiffre par un autre est faisable, mais travailler avec l'infinité de décimales de π semble autrement plus difficile !

Toutefois, il n'est pas nécessaire d'être limité en place. Après tout, même si chaque μm^2 du disque dur de mon ordinateur comporte un bit, il me suffit d'acheter autant de disques durs que nécessaire pour avoir toute la place dont je pourrais un jour avoir besoin. Cependant, la machine que conçoit Turing a une autre limitation : si on souhaite utiliser des règles pour interagir avec les données, là encore on ne peut pas faire n'importe quoi. En effet, on souhaite que notre calcul ait une description **finie**.

Ce sont là les principales raisons qui ont poussé Alan Turing à proposer le modèle ci-dessus : la dynamique (c'est-à-dire les évolutions) de la machine est décrite de manière finie, et la machine elle-même n'a accès qu'à un nombre fini

de données locales. Mais elle peut toujours aller lire ou écrire des informations arbitrairement loin, tout en sachant que ça risque alors de prendre du temps…

1.3 D'autres machines de Turing

Nous avons parlé ci-dessus de machines **déterministes**. Cependant, parfois, on ne contrôle pas tous les agissements d'une machine. Par exemple, les actions d'une station météorologique dépendent fortement de phénomènes sur lesquels on n'a aucun contrôle, et à propos desquels on n'est même pas nécessairement capable de calculer des probabilités. Dans ce contexte, quid de machines non déterministes ?

Les machines de Turing **non déterministes** sont une variante des machines déterministes, dont la table de transition est en fait une fonction $\mathcal{T} : \mathcal{A}^k \times \mathcal{S} \mapsto 2^{\mathcal{A}^k \times \{-1,0,1\}^k \times \mathcal{S}}$ (la notation 2^X représente l'ensemble des parties de X, c'est-à-dire l'ensemble des sous-ensemble de X). Dans ce cas, il existe a priori de multiples $(2k+1)$-uplets $(\mu_1, \ldots, \mu_k, \varepsilon_1, \ldots, \varepsilon_k, \mathbf{s}') \in \mathcal{T}(\lambda_1, \ldots, \lambda_k, \mathbf{s})$: on en choisit un d'une manière qui n'est pas précisée.

Puis l'on demande à la tête située sur le i-ème ruban d'écrire le symbole μ_i (effaçant ainsi le symbole λ_i qu'elle vient de lire) puis de se déplacer de ε_i cases vers la droite (donc d'une case vers la gauche si $\varepsilon_i = -1$) ; la machine rentre alors dans l'état $\mathbf{s}'$.

En particulier, notons que toute machine déterministe est en fait une machine non déterministe telle que chaque ensemble $\mathcal{T}(\lambda_1, \ldots, \lambda_k, \mathbf{s})$ est un singleton. Dans une machine non déterministe, on a identifié un ensemble de comportements possibles, et on n'essaie pas (a priori) de savoir si l'un va être plus vraisemblable qu'un autre.

Cependant, on ne peut pas faire n'importe quoi non plus : en effet, dans de nombreux cas, on peut vouloir reposer sur la chance pour mener à bien un algorithme, mais on n'en espère pas moins des propriétés de **cohérence**, c'est-à-dire que l'on s'attend à trouver le même résultat final quels que soient les choix aléatoires que l'on aura pu faire en cours de route. Ainsi, si une machine de Turing, sur une même entrée, termine deux exécutions différentes de son calcul, on demande que les deux calculs aboutissent au même résultat, que l'on appellera donc « résultat des calculs de la machine du Turing ». En revanche, habituellement, on tolère le fait que la terminaison du calcul puisse dépendre de la chance : si au moins un calcul se termine, on a un (unique) résultat, et cela nous suffit !

Ce type de machines est donc assez générique : d'une part, il englobe les

machines de Turing déterministes. D'autre part, il nous laisse toute latitue pour déterminer plus finement la façon dont on pourrait effectuer nos choix de $(2k+1)$-uplets. En outre, concrètement, à quoi peut bien servir une telle machine ? Et bien, tous les jours, vous utilisez le non-déterminisme. Par exemple, en vous fondant sur votre intuition, sur des analogies que vous aurez pu relever. Avant d'avoir fini votre raisonnement, vous ne savez pas s'il va aboutir, Mais vous tentez quand même d'avancer. Si vous avez de la chance, vous finirez par trouver la réponse au problème ; si vous n'en avez pas, vous risquez de chercher longtemps, voire de ne jamais trouver de réponse. Cependant, dans tous les cas, si vous trouvez une réponse, alors il doit s'agir de la réponse au problème, et vous ne voulez pas que cette réponse dépende du chemin que vous aurez choisi dans votre quête mathématique.

De même, les machines de Turing non déterministes sont là pour laisser sa place à la chance, voire à un **orcale** : un oracle va être une source que l'on ne contrôle pas, à qui on fait modérément confiance, mais qui prétend nous donner une solution à notre problème. Usuellement, un oracle sera un élève qui propose sa solution à un problème d'olympiade, et sa solution peut être vraie ou fausse. Si cette solution est correcte, alors le correcteur aura pour tâche de la valider ; si elle est incorrecte, il ne la validera pas. Cette capacité à vérifier une solution que l'on ne pourrait pas soi-même trouver est très importante (par exemple factoriser un grand nombre est censé être difficile, mais vérifier une factorisation est facile), sera revue dans la suite de ce cours, et fait essentiellement appel aux machines non déterministes.

Exercice 1 Soit **M** une machine de Turing non déterministe à k rubans, et **D** une configuration de départ des k rubans. Montrer que, si tous les calculs de la machine **M** sur la configuration **D** se terminent, alors il existe un nombre fini de tels calculs.

Indication : on pourra modéliser l'ensemble des calculs possibles par un arbre puis utiliser le principe des tiroirs.

D'autre part, et outre les machines de Turing non déterministes, on pourrait concevoir d'autres modèles, par exemple des machines **probabilistes**, qui nous permettraient de quantifier la probabilité d'occurrence d'un résultat : à chaque étape, on aurait une probabilité **p** de choisir tel ou tel élément de l'ensemble $\mathcal{T}(\lambda_1, \ldots, \lambda_k, \mathbf{s})$. De tels modèles existent bel et bien et sont tout à fait utilisables en pratique.

Cependant, usuellement, on préfère les simuler à l'aide de machines de Turing déterministes telles que celles vues plus haut. On se contente alors de

rajouter ℓ ruban (où ℓ est un entier a priori arbitraire) jouant rôle spécifique : chaque case a une probabilité $\frac{1}{2}$ de contenir le symbole $\times$ et une probabilité $\frac{1}{2}$ de contenir le symbole $\circ$ (de sorte que, a priori, on peut avoir une infinité de symboles $\times$), ces probabilités étant mutuellement indépendantes ; en outre, à chaque étape, la tête de lecture et d'écriture située sur un tel ruban est **obligée** de se déplacer d'une case vers la droite, de sorte que chaque symbole sera lu au plus une fois.

1.4 Fonctions calculables

À chaque étape d'un calcul (notamment au début et à la fin du calcul), une machine de Turing (déterministe) travaille sur une bande dont un nombre fini de cases a une importance ; chacune de ces cases contient un symbole choisi parmi un ensemble fini. On peut donc voir une machine de Turing comme une fonction **partielle** qui, à certaines configurations initiales des k rubans, associe une configuration finale de ces mêmes k rubans.

Pourquoi une fonction partielle ? Tout simplement parce que l'exécution de la machine de Turing ne se termine pas nécessairement.

Cependant, pour plus de praticité, on rechigne à travailler sur des fonctions dont l'argument et la valeur de sortie sont des suites de rubans finiment remplis. On va donc restreindre un petit peu notre cadre d'étude, et faire jouer aux rubans des rôles particuliers. Dorénavant, au lieu de parler d'une machine à k rubans, on on parlera d'une machine à ℓ_1 rubans **d'entrée**, à ℓ_2 rubans **de sortie** et à ℓ_3 rubans **intermédiaires** : concrètement, une telle machine représente une fonction à ℓ_1 arguments qui renvoie un ℓ_2-uplet, et qui peut utiliser ℓ_3 rubans additionnels pour les besoins de ses propres calculs. En outre, on supposera qu'elle n'a pas le droit de modifier ses rubans d'entrée, tandis que ses rubans de sortie sont initialement de la forme $\times \circ \circ \cdots$ On pourra alors dire qu'une fonction est calculable s'il existe une machine de Turing qui lui est associée.

En outre, et toujours dans le but de se rapprocher de ce que l'on connaît usuellement, on souhaite considérer des fonctions dont les arguments et les valeurs de retour sont des entiers, non des rubans. Pour ce faire, on a deux manières d'identifier un ruban à un entier :

- en base 1, on identifie l'entier n au ruban $\times\ 1\ 1\ \cdots\ 1\ \circ\ \circ \cdots$ avec n symboles « 1 » consécutifs ;
- en base 2, on identifie l'entier $n = \overline{b_k b_{k-1} \cdots b_0}^2$ au ruban $\times\ b_0\ b_1\ \cdots\ b_k \circ \circ \cdots$

La notions de fonction calculable se transpose alors instantanément en fonc-

tions partielles $f : \mathbb{N}^{\ell_1} \mapsto \mathbb{N}^{\ell_2}$. Plus précisément, soit $f : \mathbb{N}^{\ell_1} \mapsto \mathbb{N}^{\ell_2}$ une fonction partielle, et soit $\mathcal{D}_f$ son domaine de définition. On dit que f est représentée par une machine de Turing $\mathbf{M}_f$ à ℓ_1 rubans d'entrée et à ℓ_2 rubans de sortie (on ne s'intéresse pas au nombre de rubans intermédiaires) si :

- pour tout ℓ_1-uplet d'entiers $(n_1, \ldots, n_{\ell_1}) \in \mathcal{D}_f$, l'exécution de $\mathbf{M}_f$ à partir de rubans d'entrée représentant le ℓ_1-uplet $(n_1, \ldots, n_{\ell_1})$ se termine, et aboutit à une configuration où les rubans de sortie représentent le ℓ_2-uplet $f(n_1, \ldots, n_{\ell_1})$;
- pour tout ℓ_1-uplet d'entiers $(n_1, \ldots, n_{\ell_1}) \notin \mathcal{D}_f$, l'exécution de $\mathbf{M}_f$ à partir de rubans d'entrée représentant le ℓ_1-uplet $(n_1, \ldots, n_{\ell_1})$ ne se termine jamais.

On peut alors vérifier que de nombreuses fonctions usuelles (vous pourrez considérer la base que vous voulez).

Exercice 1 Les fonctions $f_1 : n \mapsto 0$, $f_2 : n \mapsto n + 1$, $f_3 : n \mapsto \max\{0, n - 1\}$, $f_4 : n \mapsto n \pmod 2$, $f_5 : n \mapsto 2n$ et $f_6 : (m, n) \mapsto \mathbf{1}_{m \leqslant n}$ sont-elles calculables ?

Mais pourquoi dire « fonction calculable » sans mention de la base considérée ? Tout simplement car on peut passer d'une base à l'autre, en vertu de l'exercice suivant :

Exercice 2 Montrer qu'il existe une machine de Turing qui, à tout ruban $\times\, 1\, 1\, \cdots\, 1 \circ \circ \cdots$ avec $n = \overline{b_k b_{k-1} \cdots b_0}^2$ symboles « 1 » consécutifs, associe le ruban $\times\, b_0\, b_1\, \cdots\, b_k \circ \circ \cdots$

Montrer qu'il existe une (autre) machine de Turing qui, à tout ruban $\times\, b_0\, b_1\, \cdots\, b_k \circ \circ$ associe le ruban $\times\, 1\, 1\, \cdots\, 1 \circ \circ \cdots$ avec $n = \overline{b_k b_{k-1} \cdots b_0}^2$ symboles « 1 » consécutifs.

On va maintenant chercher à construire, de manière générique, toute une famille de fonctions calculables.

Exercice 3 Soit p_i une projection, c'est-à-dire une fonction de la forme $p_i : (x_1, \ldots, x_n) \mapsto x_i$. Montrer que p_i est calculable.

Exercice 4 Soit $g_1, \ldots, g_m : \mathbb{N}^m \mapsto \mathbb{N}$ et $h : \mathbb{N}^m \mapsto \mathbb{N}$ des fonctions calculables. Montrer que la composée $(x_1, \ldots, x_n) \mapsto h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ est calculable.

Exercice 5 Soit $g : \mathbb{N}^{n-1} \mapsto \mathbb{N}$ et $h : \mathbb{N}^{n+1} \mapsto \mathbb{N}$ des fonctions calculables. Montrer que la fonction f définie inductivement par $f : (x_1, \ldots, x_{n-1}, 0) \mapsto g(x_1, \ldots, x_{n-1})$ et par $f : (x_1, \ldots, x_{n-1}, y+1) \mapsto h(x_1, \ldots, x_{n-1}, y, f(x_1, \ldots, x_{n-1}, y))$ est calculable.

Exercice 6 Soit $g : \mathbb{N}^{n+1} \mapsto \mathbb{N}$. Montrer que la fonction f **partiellement** définie par $f : (x_1, \ldots, x_n) \mapsto \min\{k \geqslant 0 : g(x_1, \ldots, x_n, k) = 0\}$ est calculable.

Notons que c'est bien ce dernier point qui montre que l'on doit considérer des fonctions **partielles** $f : \mathbb{N}^{\ell_1} \mapsto \mathbb{N}^{\ell_2}$ et non pas uniquement des fonctions totales.

Exercice 7 Montrer qu'il existe une machine de Turing représentant la fonction partielle f, dont le domaine de définition est $2\mathbb{N}$, et telle que $f : n \mapsto 0$ si n est pair.

En outre, on peut en fait se rendre compte que considérer des machines de Turing non déterministes ne modifie pas la notion de « fonction calculable » !

Exercice 8 Montrer que, pour toute machine de Turing non déterministe à ℓ_1 rubans d'entrée et ℓ_2 rubans de sortie, il existe une machine de Turing déterministe (à ℓ_1 rubans d'entrée et ℓ_2 rubans de sortie) représentant la même fonction partielle.

1.5 Et le lien avec les langages de programmation ?

Ce modèle est bien sympathique et semble assez puissant, mais est-il en accord avec ce que l'on peut rêver de programmer sur une vraie machine ? Tout d'abord, les intuitions de Turing données précédemment semblent bien indiquer que oui. Mais on peut essayer de s'en convaincre d'une autre manière.

Considérons un programme informatique. Afin de faciliter la clarté de l'exposition, on va supposer que toutes nos instructions sont « atomiques », c'est-à-dire qu'on n'imbrique pas deux instructions en une seule. On obtient alors un programme du type :

```
Argument : entier n ⩾ 0;
i ← n;
c ← 0;
while i ≠ 1 do
    j ← i;
    ℓ ← 0;
    while j > 1 do
        ℓ ← ℓ + 1;
        j ← j − 2;
    end
    if j = 0 then
        i ← ℓ;
    else
        i ← 3 × i;
        i ← i + 1;
    end
    c ← c + 1;
end
Renvoyer l'entier c;
```

Un programme comporte plusieurs types d'instructions :
- les **assignations de valeurs**, telles que $i \leftarrow \ell$ ou $j \leftarrow j - 2$,
- les **opérations arithmétiques**, telles que le calcul de $j - 2$ ou de $3 \times i$,
- les **tests**, tels que **if** $j = 0$ **then** $\cdots$ **else** $\cdots$ **end**, et
- les **boucles while**, telles que **while** $j > 1$ **do** $\cdots$ **end**.

Notons d'ailleurs que l'on pourrait aussi utiliser des **boucles for**, mais que l'on peut remplacer celles-ci par des **boucles while**.

On pourrait alors montrer, par récurrence sur la longueur du programme (par exemple le nombre de caractères utilisés) que chaque programme ainsi créé représente en fait une fonction calculable.

Exercice 1 Montrer que que tout programme ainsi créé représente effectivement une fonction calculable.

Indication : On pourra s'appuyer sur les exercices précédents.

1.6 Théorème d'incomplétude de Gödel

On va ici présenter une variante du théorème d'incomplétude de Gödel :

Théorème 1. Il existe un théorème vrai que l'on ne pourra jamais démontrer.

Évidemment, il faut ruser, vu qu'on aura du mal à donner un exemple explicite d'un tel théorème (pourquoi ?). On va donc procéder en plusieurs étapes.

Exercice 1 Montrer que l'ensemble des fonctions totales $f : \mathbb{N} \mapsto \mathbb{N}$ calculables est en bijection avec $\mathbb{N}$.

On peut d'ores et déjà montrer qu'il existe des fonctions non calculables !

Exercice 2 Montrer qu'il existe des fonctions non calculables.
Indication : On pourra s'appuyer sur le cours sur la dénombrabilité.

Mais on peut également construire explicitement une fonction non calculable : puisque l'on a montré l'existence une telle bijection ψ entre $\mathbb{N}$ et $\{f : \mathbb{N} \mapsto \mathbb{N} : f \text{ est totale et calculable}\}$, servons-nous en !

Exercice 3 Construire une fonction « trop grande pour être calculable », c'est-à-dire une fonction F telle que, pour toute fonction calculable f, on ait $\lim_{n\to\infty} F(n) - f(n) = +\infty$.

C'est là une fonction qui croît très vite ! En particulier, on peut encore s'approcher du théorème d'incomplétude de Gödel.

Exercice 4 Montrer que la bijection ψ n'est pas calculable, puis qu'il n'existe aucun algorithme qui, étant donné une machine de Turing à un ruban d'entrée et un ruban de sortie, indique si elle représente une fonction totale.

On n'a donc pas de méthode algorithmique systématique nous permettant de vérifier si une machine de Turing représente une fonction totale. Or, de même que l'ensemble $\{f : \mathbb{N} \mapsto \mathbb{N} : f \text{ est totale et calculable}\}$ est en bijection avec $\mathbb{N}$, l'ensemble des démonstrations est également en bijection avec $\mathbb{N}$. De surcroît, par essence, on peut calculer une telle bijection : il suffit d'énumérer toutes les suites de 1 caractère et de tester celles qui sont une démonstration (vraisemblablement, pas beaucoup), puis de recommencer avec les suites de 2 caractères, et ainsi de suite. On aboutit donc au résultat suivant.

Théorème 2. Il existe une méthode algorithmique systématique pour énumérer toutes les preuves de tous les théorèmes. [2]

On peut donc conclure sur un cas particulier du théorème de Gödel présenté précédemment.

2. Cette méthode permet donc notamment de résoudre tous les problèmes d'olympiades ☺.

Théorème 3. Il existe une machine de Turing dont on ne peut ni montrer qu'elle représente une fonction totale, ni montrer qu'elle ne représente pas une fonction totale.

Exercice 5 Se convaincre du théorème précédent.

2 Complexité

On vient de voir ci-dessus qu'il existait une méthode permettant de résoudre tous les problèmes d'olympiades. Si on ne vous l'a pas enseignée, c'est sans doute que cette méthode souffre d'un affreux défaut : elle est extrêmement lente. Au contraire, la méthode consistant à suivre en cours et à s'entraîner assidûment à faire des mathématiques amusantes est redoutablement plus efficace. On perçoit donc que, outre la notion de **calculabilité**, on a également besoin d'une notion de **complexité** : est-il difficile d'aboutir à tel ou tel résultat ?

2.1 Différentes mesures de complexité

De manière usuelle, on considère souvent deux mesures de la complexité, sur lesquelles on se focalisera dans la suite de ce cours : la complexité en **espace** et la complexité en **temps**. La première mesure concerne la place mémoire nécessaire (c'est-à-dire peu ou prou le nombre maximum de cases utilisées sur nos k rubans) pour mener à bien un calcul, tandis que la seconde concerne le nombre d'opérations nécessaires (si chaque opération dure un temps constant, alors effectuer 10 fois plus d'opérations prend 10 fois plus de temps).

On distingue déjà une limite de ces mesures : par exemple, qui a dit qu'une opération prenait un temps constant ? Après tout, écrire un bit en mémoire prend moins de temps que lancer une requête sur internet. En outre, il serait tout-à-fait concevable de paralléliser les calculs : on va plus vite si 100 personnes réalisent chacune 100 opérations plutôt que si 1 personne réalise 1000 opérations toute seule…

D'autre part, la complexité d'un algorithme dépend évidemment de l'argument qu'on lui fournit : ainsi, un algorithme pour calculer la factorielle va vite quand il s'agit de calculer 0!, mais risque d'aller moins vite dès lors que l'on s'intéresse à 6!!!. Cela montre bien que la complexité d'un algorithme est en fait une **fonction** (éventuellement non calculable, d'ailleurs) qui, à chaque

entrée, associe les coûts, en mémoire, temps ou autres, de l'exécution de l'algorithme sur cette entrée.

Cependant, ce faisant, on se retrouve avec des informations parfois peu pratiques : en effet, a priori, la complexité d'un algorithme dépend de chaque entrée, et calculer cette complexité pour chaque entrée possible demanderait beaucoup trop d'efforts en pratique ! Aussi les théoriciens de la complexité ont-ils eu l'idée géniale de sacrifier la précision à l'efficacité, et de chercher à calculer la complexité d'un algorithme non plus en fonction de l'entrée elle-même, mais uniquement en fonction de certaines caractéristiques de l'entrée.

Par exemple, considérons la machine de Turing $\mathbf{M}$ suivante :

- $\mathbf{M}$ travaille avec 1 ruban de lecture, 1 ruban d'écriture et 0 ruban additionel ;
- l'alphabet est $\mathcal{A} = \{\times, 0, 1, \circ\}$;
- l'ensemble des états est $\mathcal{S} = \{\mathbf{s_i}, \mathbf{s_f}\}$;
- la table de transition est $\mathcal{T} : (\circ, \lambda_2, \sigma) \mapsto (\circ, 1, 1, 1, \mathbf{s_f})$ et $\mathcal{T} : (\lambda_1, \lambda_2, \sigma) \mapsto (\lambda_1, \lambda_2, 1, 1, \sigma)$ pour tous $\lambda_1 \in \{\times, 1\}$, $\lambda_1 \in \mathcal{A}$ et $\sigma \in \mathcal{S}$.

Que fait cette machine de Turing (qui travaille manifestement sur des entiers représentés en base 2) ? Elle représente la fonction $f : n \mapsto n + 2^m$, où m est le plus petit entier naturel tel que $2^m > n$. En outre, si n a k chiffres en base 2, alors $\mathbf{M}$ va écrire sur $k + 2$ cases, et va utiliser $k + 2$ étapes. On constate donc que le paramètre qui détermine vraiment les complexités spatiale et temporelle de notre algorithme est k, et non pas n lui-même.

De manière générale, on prendra souvent pour mesure de la complexité d'une donnée d'entrée le nombre de cases nécessaires pour la représenter (en ignorant les $\circ$ situés à droite de chaque bande). Dans ce cas-là, on constate que, sur une entrée $\mathbf{e}$ de complexité $\|\mathbf{e}\|$, l'algorithme utilisé est de complexités spatiale et temporelle $\|\mathbf{e}\| + 1$. De surcroît, utiliser cette mesure de complexité a l'avantage de faciliter l'utilisation en chaîne de plusieurs machines de Turing.

Par exemple, supposons qu'une machine $\mathbf{M_a}$, lisant une entrée qui prend une taille $\mathbf{t}$, effectue $\mathbf{t}^2$ opérations, écrit nécessairement sur $3\mathbf{t}$ cases et renvoie nécessairement un résultat qui tient sur $\mathbf{t} + 1$ cases ; une autre machine $\mathbf{M_b}$, lisant une entrée qui prend une taille $\mathbf{u}$, effectue $2^{\mathbf{u}}$ opérations, écrit nécessairement sur $\mathbf{u}^3$ cases et renvoie nécessairement un résultat qui tient sur ces $\mathbf{u}^3$ cases. Et bien en utilisant la machine $\mathbf{M_a}$, sur une entrée de taille $\mathbf{t}$ puis la machine $\mathbf{M_b}$ sur le résultat obtenu, on aura effectué $\mathbf{t}^2 + 2^{\mathbf{t}+1}$ opérations, écrit sur $3\mathbf{t} + (\mathbf{t} + 1)^3$ cases, et renvoyé un résultat qui tient sur $(\mathbf{t} + 1)^3$ cases.

2.2 Complexité polynomiale

Dorénavant, on va essentiellement travailler avec les mesures de complexités mentionnées ci-dessus : la « complexité » d'un ℓ-uplet d'entiers $(n_1, \ldots, n_\ell)$ sera tout simplement le nombre de cases nécessaires pour représenter ce ℓ-uplet, c'est-à-dire $2\ell + \sum_{i=1}^{\ell} \max\{0, \lceil \log_2(n_i) \rceil\}$. Pour des raisons pratiques, on notera ci-dessous ce ℓ-uplet par $\mathbf{n}$, et sa complexité par $\|\mathbf{n}\|$.

On peut alors étudier la complexité spatiale ou temporelle d'un calcul en fonction de $\mathbf{n}$. Puis on peut exprimer la complexité d'une machine de Turing $\mathbf{M}$, déterministe ou non, en fonction de son ℓ-uplet d'entrée $\mathbf{n}$: ce sera la plus petite complexité de tous les calculs finis exécutables à partir de l'entrée $\mathbf{n}$. Si $\mathbf{M}$ est déterministe, alors il existe au plus un unique tel calcul, mais de toute manière la plus petite complexité existe bel et bien dès lors qu'au moins un calcul se termine. Si aucun calcul de $\mathbf{M}$ ne se termine quand l'entrée est $\mathbf{n}$, la complexité de $\mathbf{M}$ sur l'entrée $\mathbf{n}$ n'est tout simplement pas définie. En effet, on pourrait légitimement considérer que la complexité temporelle devrait être infinie, mais rien n'empêche $\mathbf{M}$ de boucler bêtement, effectuant une infinité d'opérations sans jamais écrire quoi que ce soit, de sorte qu'il ne serait guère aisé de considérer la complexité spatiale d'un calcul qui n'aboutit pas…

Puis l'on va dire que la machine $\mathbf{M}$ fonctionne en temps (ou en espace) polynomial s'il existe un polynôme $P \in \mathbb{R}[X]$ tel que, quelle que soit l'entrée $\mathbf{n}$ pour laquelle $\mathbf{M}$ a une complexité temporelle (ou spatiale), cette complexité sera au plus $P(\|\mathbf{n}\|)$. Pourquoi polynomial ?

Exercice 1 Montrer que si la machine $\mathbf{M}$ fonctionnent en temps polynomial, alors elle fonctionne également en espace polynomial.

Exercice 2 Montrer que si $\mathbf{M}$ et $\mathbf{M}'$ sont deux machines de Turing qui fonctionnent en temps (ou en espace) polynomial, alors exécuter $\mathbf{M}$ puis exécuter $\mathbf{M}'$ sur le résultat de $\mathbf{M}$ revient à appliquer une machine de Turing qui fonctionne en temps (ou en espace polynomial).

2.3 Problèmes de décision — P = NP ?

Parmi les fonctions les plus communément utilisées figurent les fonctions **indicatrices**. Si, si, vous vous en servez tous les jours, à chaque fois que vous vous demandez « est-ce que …est démontrable ? ». En effet, quitte à identifier chaque proposition mathématique à un entier naturel (par exemple en énumérant les propositions mathématiques, comme on l'a déjà fait un certain nombre de fois), cette question revient à calculer $\mathbf{1}_{\mathbf{D}}(\mathbf{p})$, où $\mathbf{p}$ est le numéro de

la proposition dont on se demande si elle est démontrable, et **D** est l'ensemble des numéros de toutes les propositions démontrables.

Un **problème de décision** est donc un problème tel que celui-ci : étant donné un ensemble $E \subseteq \mathbb{N}^\ell$, on cherche à décider si le ℓ-uplet en entrée appartient bien à E. On dira alors que est **décidable** ou **calculable** si la fonction indicatrice $\mathbf{1_E} : \mathbb{N}^\ell \mapsto \mathbb{N}$ est calculable. Cependant, si $\mathbf{1_E}$ n'est pas calculable, il ne faut pas perdre espoir tout de suite !

Avec un peu de chance, notre ensemble E sera quand même **semi-décidable**, c'est-à-dire qu'il existe une fonction partielle calculable $f : \mathbb{N}^\ell \mapsto \mathbb{N}$, de domaine de définition un ensemble $\mathcal{D}_f$ tel que $E \subseteq \mathcal{D}_f \subseteq \mathbb{N}^\ell$, et telle que $f(a_1, \ldots, a_\ell) = \mathbf{1_E}(a_1, \ldots, a_\ell)$ pour tout ℓ-uplet $(a_1, \ldots, a_\ell) \in \mathcal{D}_f$. On dira alors que tout algorithme permettant de calculer f **semi-décide** l'ensemble E. En somme, E est un ensemble décidable si, pour tout ℓ-uplet d'entiers, on peut décider s'il appartient à E ou pas ; E est semi-décidable si, pour tout ℓ-uplet d'entiers appartenant à E, on peut vérifier qu'il appartient bien à E.

Exercice 1 Montrer que, si E est un ensemble semi-décidable, alors la fonction partielle f dont E est l'ensemble de définition et telle que $f : x \mapsto 1$ quand $x \in E$ est une fonction calculable.

Maintenant que l'on a vu qu'un problème de décision n'était rien d'autre que le problème du calcul d'une fonction indicatrice, les notions de complexité vues précédemment s'appliquent directement ! Cela va nous permttre d'introduire des **classes de complexité** que sont **P** et **NP**.

Définition 4. La classe **P** est la classe des ensembles **semi-décidables en temps polynomial**. Plus précisément, un ensemble E est dans la classe **P** s'il existe une machine de Turing **M** déterministe qui semi-décide l'ensemble E en temps polynomial.

La classe **NP** est la classe des ensembles **semi-décidables en temps polynomial non déterministe**. Plus précisément, un ensemble E est dans la classe **NP** s'il existe une machine de Turing **M** non déterministe qui semi-décide l'ensemble E en temps polynomial.

Il est clair que la classe **P** est une sous-classe de la classe **NP**. La question cruciale qu'est **P = NP ?** consiste donc à se demander si les deux classes **P** et **NP** sont égales (et non pas, comme me l'ont déjà signalé certains mauvais esprits, à décider si $\mathbf{P} = 0$ ou $\mathbf{N} = 1$). Cette question est apparemment très dure, vu que de multiples esprits parmi les plus brillants s'y sont cassé les dents. Néanmoins, on peut d'ores et déjà obtenir certains résultats sympathiques.

Exercice 2 Montrer que la classe **P** est close par passage au complémentaire, c'est-à-dire que si E est une partie de $\mathbb{N}^\ell$ telle que si $E \in \mathbf{P}$, alors $\mathbb{N}^\ell \setminus E \in \mathbf{P}$ également.

On vient de montrer, grâce à cet exercice, que **P** est en fait la classe des ensembles dont le test d'appartenance est carrément **décidable** (et pas seulement semi-décidable) en temps polynomial : intuitivement, un ensemble E appartient à la classe **P** s'il est raisonnablement faisable de décider l'appartenance à E. En revanche, la classe **NP** est plus subtile : un ensemble E appartient à la classe **NP** s'il est possible, en ayant une chance inouïe, de vérifier que les éléments de E appartiennent bien à E ; ou, autrement dit, si, pour chaque élément $e \in E$, il existe une preuve lisible en temps raisonnable du fait que $e \in E$.

Tout compte fait, la classe **NP** est celle sur laquelle les mathématiciens travaillent toute la journée : il s'agit de problèmes ayant une solution lisible mais sans doute difficile à trouver. Heureusement pour nos amis matheux, une solution « difficile à trouver » est, par définition, « trouvable », comme on peut effectivement le vérifier.

En outre, tout un pan de l'informatique repose sur des variantes de l'hypothèse selon laquelle $\mathbf{P} \neq \mathbf{NP}$: la **cryptographie**. Un exemple (qu'il ne faut cependant jamais utiliser tel quel) est celui du système de chiffrement RSA.

Exemple 5. Pour pouvoir recevoir secrètement des messages du monde entier, Alice a choisi deux grands nombres premiers distincts p et q, ainsi que deux entiers d et e tels que $de \equiv 1 \pmod{(p-1)(q-1)}$. Elle rend alors publics (par un moyen que l'on n'explicitera pas ici) la paire d'entiers (pq, d).

Puis, si un jour Bob veut envoyer un message secret à Alice, il se débrouille pour coder son message sous forme d'éléments de l'ensemble $\{0, 1, \ldots, pq - 1\}$, de sorte qu'il lui suffit d'envoyer secrètement à Alice des résidus modulo pq. Au lieu d'envoyer l'entier M directement, il envoie alors l'entier $M^d \pmod{pq}$. Puis Alice pourra calculer $M \equiv (M^d)^e \pmod{pq}$, et ainsi retrouver le message de Bob : c'est le chiffrement RSA.

Exercice 3 Montrer que l'on a effectivement $M \equiv (M^d)^e \pmod{pq}$.

Quel est le lien avec l'hypothèse selon laquelle $\mathbf{P} \neq \mathbf{NP}$? Et bien on aimerait bien que connaître pq, d et M^d ne soit pas suffisant pour pouvoir calculer M de manière efficace ; autrement dit, que le calcul de M **ne puisse pas se faire en temps polynomial** par qui connaît uniquement pq, d et M^d. En revanche, il doit bien évidemment être possible de vérifier efficacement que M est le message envoyé par Bob, vu que cette vérification est déjà effectuée par Bob

lui-même : il suffit de calculer M^d ; ainsi le calcul de M **doit pouvoir se faire en temps polynomial**. Voilà pourquoi on souhaiterait que le calcul de M à partir de pq, d et M^d soit dans **NP** mais pas dans **P**.

Le lecteur attentif aura cependant noté que la réalité est en fait légèrement plus subtile, puisque l'on parlait plus haut de problèmes de décision, alors que l'on parle ici de problèmes de calcul. Cependant, les deux situations sont en fait très analogues : en effet, être capable de calculer un entier dont on connaît la taille, c'est être capable de décider si chacun de ses chiffres binaires est un zéro…

Exercice 4 Montrer que la classe **NP** est incluse dans la classe des ensembles décidables, c'est-à-dire que si E est une partie de $\mathbb{N}^\ell$ telle que si $E \in$ **NP**, alors E est un ensemble décidable.

Pour autant, les classes **P** est **NP** contiennent toutes les deux des problèmes intéressants !

Exercice 5 Montrer que l'ensemble $\{(a, b, n, d) : a^b \equiv d \pmod{n}\}$ appartient à la classe **P**.

Exercice 6 Montrer que l'ensemble des nombres premiers et que l'ensemble des nombres composés appartiennent tous deux à la classe **NP**.