

1 Le modèle de calcul

Le propos de la complexité algorithmique est de mesurer la difficulté d'un problème à l'aune de l'efficacité des algorithmes pour le résoudre. Dans ce premier chapitre, nous allons définir formellement ce que nous entendons par problèmes et algorithmes.

Aujourd'hui nous avons tous une idée intuitive de ce qu'est un algorithme car nous manipulons chaque jour des ordinateurs ; toutefois, il n'est pas si aisé d'en donner une définition formelle. La définition habituelle que nous allons suivre ici passe par les machines de Turing. Il est remarquable que ce modèle de calcul ait été proposé par Turing dès 1937 dans son article [Tur37], *avant* la naissance des ordinateurs, à laquelle il a d'ailleurs largement contribué.

La définition de Turing est encore utilisée actuellement car, afin d'étudier les algorithmes, on souhaite un modèle théorique aussi simple que possible, d'une part, et d'autre part nous verrons que ce modèle rudimentaire reste capable d'effectuer les mêmes tâches que nos ordinateurs. À la même époque, d'autres formalismes équivalents ont été introduits (λ -calcul par Church [Chu36], fonctions récursives par Herbrand et Gödel, etc.) mais la machine de Turing a l'avantage de permettre la définition aisée du temps et de l'espace utilisés lors d'un calcul. Des modèles plus proches de nos ordinateurs existent également (machines RAM par exemple) mais, pour étudier la complexité algorithmique, la connaissance des machines de Turing est devenue trop usuelle pour pouvoir s'en passer — bien que ce ne soit pas toujours le modèle idéal.¹

Une telle machine manipule des suites finies de symboles, c'est-à-dire qu'on représente par des « mots » les données à traiter (nous aurons donc besoin d'un codage des données en des mots). Elle effectue alors un calcul sur un mot qu'on lui fournit en entrée et renvoie le résultat du calcul sous la forme d'un nouveau mot de sortie comme illustré à la figure 1.1.

1. On remarquera également qu'il est malaisé d'employer un autre modèle de calcul pour obtenir exactement la définition habituelle des concepts que nous aborderons dans ce livre.

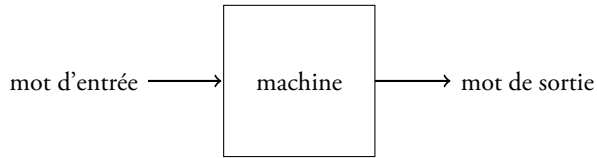


FIGURE 1.1 – Principe d’une machine de Turing.

Nous allons d’abord nous attarder sur la notion de problème avant de définir la machine de Turing et de voir qu’une telle machine permet bien de réaliser toutes les opérations qu’on souhaite possibles pour un algorithme.

1.1 Problèmes, langages et codage

1.1.1 Codage

Chacun sait que les ordinateurs actuels ne manipulent que des 0 (absence de courant électrique) et des 1 (présence du courant). Et pourtant, dans nos programmes nous pouvons parler d’objets de diverse nature : des entiers, des rationnels, des matrices, des graphes, etc. Pour cela, il faut *coder* ces objets par une suite de 0 et de 1.

Dans le cadre plus général de ce livre, nous ne nous contenterons pas de deux symboles 0 et 1, mais d’un *alphabet* fini quelconque. Un alphabet est simplement l’ensemble des symboles autorisés. Ainsi, l’alphabet binaire est $\{0, 1\}$ mais on pourrait aussi travailler sur l’alphabet $\{a, b, c, \dots, z\}$ par exemple, ou bien $\{a, b, \#\}$ pour parler des mots (c’est-à-dire des suites finies de symboles) composés des lettres a , b et $\#$ (le symbole $\#$ servira la plupart du temps de délimiteur). Dès lors, nous devons coder les objets à manipuler par des mots sur un alphabet fini Σ .

La taille d’un mot x , notée $|x|$, est le nombre de symboles que x contient et joue une grande importance puisque nous verrons plus tard que le nombre d’étapes de calcul d’une machine sera considéré comme une fonction de la taille du mot d’entrée. Ainsi, nous verrons par la suite que *la difficulté d’un problème dépend du codage utilisé*. Il faudrait donc toujours préciser ce codage. Néanmoins, il y a plusieurs codages usuels que nous utiliserons par défaut, sauf mention contraire :

- les entiers codés en binaire avec les symboles 0 et 1 (par exemple, l’entier 12 sera codé 1100) ;
- de même pour les rationnels, le numérateur et le dénominateur séparés par un délimiteur (par exemple, $4/3$ sera codé 100#11) ;
- une matrice donnée par la liste de ses coefficients séparés par des délimiteurs ;
- un graphe donné par sa matrice d’adjacence ;
- un polynôme donné par la liste de ses coefficients (séparés par des délimiteurs), etc.

Mais il existe également des codages moins usuels, en voici deux exemples :

- les entiers codés en unaire avec le seul symbole 1 (le nombre de 1 donne la valeur de l'entier, par exemple 5 sera encodé 11111) : bien que non usuel, nous rencontrerons ce codage plus tard ;
- un polynôme de degré d donné par la liste des valeurs qu'il prend sur les entiers de 0 à d .

Enfin, on aura régulièrement besoin de coder des couples ou des n -uplets de mots. Si x et y sont deux mots sur un alphabet Σ , on désignera par (x, y) le codage du couple (x, y) , c'est-à-dire, selon les cas :

- si l'alphabet peut être augmenté (ce qui sera le cas la plupart du temps), (x, y) désignera le mot $x\#y$ sur l'alphabet $\Sigma \cup \{\#\}$ et la taille de (x, y) est alors $|x| + |y| + 1$: par exemple, si $x = abaa$ et $y = bba$ alors le code de (x, y) est $abaa\#bba$ de taille 8 ;
- lorsque l'alphabet peut être augmenté, on peut même obtenir un codage de taille $|x| + |y|$ exactement en codant x sur un alphabet suivi de y sur un alphabet disjoint : par exemple, $(abba, caba)$ sera codé $abbaCABA$;
- si l'alphabet ne peut être augmenté, on supposera qu'il possède au moins deux symboles² que l'on notera 0 et 1 ; il y a alors au moins deux codages possibles :

- si $x = x_1 \dots x_n$ et $y = y_1 \dots y_m$ alors (x, y) désignera le mot

$$x_1 0 x_2 0 x_3 0 \dots x_{n-1} 0 x_n 1 y_1 y_2 \dots y_m,$$

c'est-à-dire qu'on intercale 0 entre les lettres de x et 1 pour séparer x et y , puis on écrit y normalement : la taille de (x, y) est alors $2|x| + |y|$,

- pour un codage plus compact, on donnera d'abord la taille de x (pour savoir où séparer les deux mots) puis la concaténation des deux mots : plus précisément, si $t = |x|$ est la taille de x codée en binaire sur $\{0, 1\}$, $t = t_1 \dots t_l$, alors (x, y) désignera le mot $t_1 0 t_2 0 \dots t_{l-1} 0 t_l 1 x y$; la taille de (x, y) est alors $|x| + |y| + 2 \lceil \log(1 + |x|) \rceil$.³

On peut aisément généraliser ces codages aux n -uplets pour $n > 2$. Selon les cas, nous utiliserons le codage le plus adapté sans forcément l'explicitier si le contexte permet de le deviner. Ainsi, la plupart du temps nous supposerons dans nos raisonnements que le codage du couple (x, y) a pour taille $|x| + |y|$.

Maintenant que nous savons coder nos objets sur un alphabet fini, voyons comment formaliser les problèmes qu'on se pose sur ces objets.

1.1.2 Problèmes et langages

Le but d'un programme ou d'un algorithme est de résoudre un problème. La plupart du temps, et nous nous placerons dans ce cadre, le problème est générique mais on le résout

2. Le cas unaire peut être traité par une injection usuelle de $\mathbb{N}^2$ dans $\mathbb{N}$ mais il ne nous intéresse pas ici.

3. Rappelons que, sauf indication contraire, tous les logarithmes utilisés seront en base 2.

sur une *instance* particulière : par exemple, on ne peut répondre au problème « déterminer si un entier est premier » que sur un entier particulier qu'on nous donnerait en entrée ; ou encore, le problème « trier par ordre croissant une liste d'entiers » n'a de sens que sur une liste d'entiers sur laquelle travailler.

Un problème est donc composé de deux éléments : une entrée (ou *instance*) et une question ou une tâche à réaliser. Les deux exemples précédents se reformulent ainsi :

1. PRIMALITÉ :

- *entrée* : un entier N ;
- *question* : N est-il premier ?

2. TRI :

- *entrée* : une liste d'entiers l ;
- *tâche* : trier l par ordre croissant.

On distingue ainsi deux types de problèmes : ceux qui consistent à répondre par oui ou par non à une question donnée (dans l'exemple précédent, déterminer si un entier est premier), qu'on appelle *problèmes de décision* ; et ceux qui consistent à produire un nouvel objet (dans l'exemple précédent, produire la nouvelle liste triée), qu'on appelle *problèmes d'évaluation*. En effet, ces derniers consistent à évaluer une fonction : dans l'exemple, il s'agit d'évaluer la fonction qui à une liste d'entiers associe la liste triée.

Pour spécifier un problème de décision, il suffit de donner les mots pour lesquelles la réponse est « oui » : un tel problème est donc caractérisé par un ensemble d'instances positives. Plus précisément, un problème de décision donné sous la forme « entrée/question » peut facilement être transformé sous la forme d'un ensemble

$$L = \{x \in \Sigma^* \mid x \text{ code une instance valide et la réponse à la question sur } x \text{ est positive}\}$$

qu'on appellera « langage ». Un problème d'évaluation est quant à lui caractérisé par la fonction qu'il calcule. On obtient alors la définition suivante.

1-A Définition

- Un *langage* (ou problème de décision) sur un alphabet Σ est un ensemble de mots sur Σ . Un langage $L \subseteq \Sigma^*$ définit le problème de décision dont les instances positives sont exactement les mots de L .
- Un problème d'évaluation sur un alphabet Σ est une fonction $f : \Sigma^* \rightarrow \Sigma^*$.

Lorsqu'un problème de décision est vu comme un langage L , la forme de l'entrée apparaît dans L par « x code une instance valide ». Il faudra donc toujours avoir à l'esprit que tester si l'entrée encode une instance valide fait partie de la complexité du problème ; mais en règle générale, ce test est très facile et rapide à réaliser et est donc omis des raisonnements.

Un peu de recul

Dans la suite, nous étudierons surtout les problèmes de décision (langages). Mais il faut garder à l'esprit qu'on peut la plupart du temps transformer un problème d'évaluation en un langage de « difficulté équivalente ».

Par exemple, pour transformer « trouver le plus petit diviseur non trivial de N », on ajoutera une entrée k pour obtenir la question « existe-t-il un diviseur non trivial de N qui soit inférieur à k ? ». Si l'on sait répondre à cette question, alors on peut trouver rapidement le plus petit diviseur non trivial de N : il suffit d'effectuer une recherche dichotomique en testant s'il existe un diviseur $\leq N/2$, puis si oui, s'il existe un diviseur $\leq N/4$ ou sinon s'il existe un diviseur $\leq 3N/4$, etc.

De manière presque équivalente, plutôt qu'un entier k on pourrait ajouter un mot a à l'entrée pour obtenir la question « existe-t-il un diviseur non trivial de N dont l'écriture binaire commence par a ? ». On effectuerait alors une recherche préfixe à la place de la recherche dichotomique.

Enfin, une autre façon de procéder est de demander le i -ème bit de la réponse. Ainsi, pour transformer « calculer la somme de deux entiers x et y » en un problème de décision, on ajoutera une entrée i pour obtenir la question « le i -ème bit de la somme $x + y$ est-il 1 ? ».

Passons maintenant à l'étude du modèle de calcul qui permet de travailler sur ces problèmes.

1.2 La machine de Turing

Il existe de nombreuses variantes de la machine de Turing : un ruban ou plusieurs, infini vers la droite seulement ou bi-infini, etc. Tous ces modèles sont équivalents mais le montrer n'est pas l'objet de ce livre (on trouvera les simulations entre modèles dans des livres de calculabilité par exemple). Nous nous contenterons donc de définir une unique version de machine de Turing, à plusieurs rubans bi-infinis.

1-B Remarque Nous avons pris le parti de définir rigoureusement ce modèle de calcul ; dès lors, les définitions sont relativement laborieuses pour des concepts intuitifs. Une approche possible est de sauter le formalisme et de s'attarder sur la description informelle et les exemples.

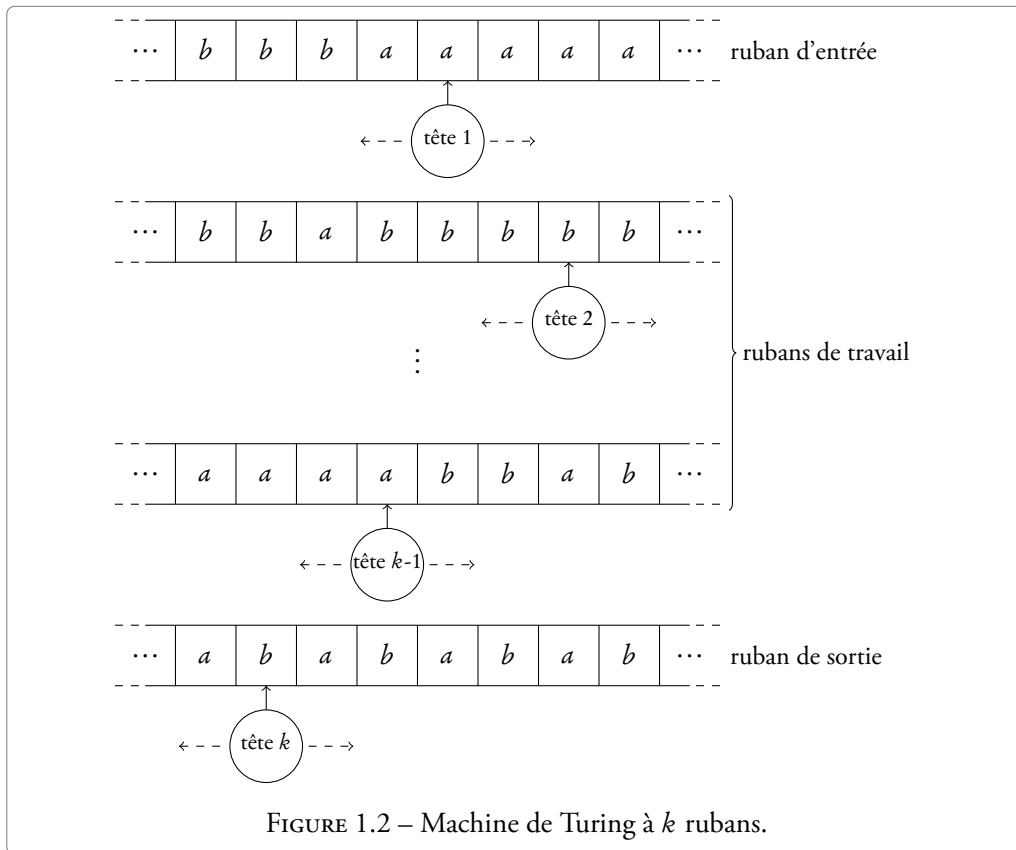
1.2.1 Définition

Une machine de Turing (figure 1.2) est composée d'un nombre fini de rubans sur chacun desquels se déplace une tête de lecture/écriture. Les rubans sont composés de cases qui contiennent chacune un symbole : il s'agit de la mémoire de notre machine. C'est l'information contenue sur ces rubans qu'on va lire et modifier. Pour cela, les têtes de

lecture/écriture peuvent se déplacer sur les rubans et lire ou modifier le contenu de la case qu'elles visitent.

On distinguera trois types de rubans⁴ :

- le ruban d'entrée, en lecture seule (la tête peut seulement lire les symboles et pas les modifier) et sur lequel est écrit le mot à traiter ;
- le ruban de sortie, en écriture seule (la tête ne peut qu'écrire des symboles sans lire le contenu des cases) et sur lequel on écrira le résultat du calcul ;
- les rubans de travail (en lecture et écriture) permettant de mener à bien le calcul.



Les têtes de lecture/écriture ont un pouvoir très limité : elles ne peuvent que se déplacer d'une case vers la gauche ou vers la droite et ont une mémoire finie, c'est-à-dire qu'elles ne peuvent prendre qu'un nombre fini d'états possibles. En d'autres termes, l'ensemble des

4. Ces conventions nous permettront de parler de manière homogène de complexité en espace ; la plupart du temps, nous aurions pu considérer des machines dont les rubans d'entrée et de sortie sont aussi en lecture/écriture.

têtes est un automate fini. Les rubans, quant à eux, sont infinis et représentent la mémoire de la machine. Si l'on indexe les cases d'un ruban par $\mathbb{Z}$, un déplacement à droite d'une tête de lecture correspond à incrémenter sa position (+1) et un déplacement à gauche à la décrémenter (−1) ; elle peut aussi rester sur place (0).

Un peu de recul

Nos ordinateurs n'ont pas une mémoire infinie : elle est certes conséquente mais finie. Ce sont donc des automates finis. Mais cette modélisation n'est pas satisfaisante : aucun programmeur n'accepterait d'utiliser un automate fini qui par essence est incapable de compter... La mémoire infinie d'une machine de Turing (ses rubans) semble refléter ce qu'un ordinateur est capable de faire et on peut évoquer deux raisons à cela :

- en pratique, la taille de la mémoire d'un ordinateur est suffisamment grande pour qu'elle soit considérée « infinie » ;
- s'il manque de mémoire pour réaliser un calcul, on peut toujours en ajouter (sous la forme d'un disque dur externe par exemple), donc la place dont on dispose est « virtuellement infinie »...

Au début du calcul, le ruban d'entrée contient le mot sur lequel la machine doit travailler. Ce mot occupe un nombre fini de cases, les autres restant vides : un symbole spécial « blanc » représente une case vide. Tous les autres rubans sont vides (c'est-à-dire ne contiennent que des symboles blancs). La tête du ruban d'entrée pointe sur la première lettre du mot d'entrée et l'état des têtes est un état particulier appelé état initial. Puis, en fonction de l'état des têtes et des symboles qu'elles lisent dans la case visitée, chaque tête remplace éventuellement par un nouveau symbole le contenu de la case, se déplace à gauche ou à droite et change d'état. Le calcul se termine lorsqu'un « état terminal » est atteint. Formalisons cette description avant de voir des exemples.

Pour définir formellement une machine de Turing, nous devons donc spécifier plusieurs éléments : les symboles utilisés sur les rubans, les états des têtes et la façon dont elles agissent sur les rubans.

1-C Définition (machine de Turing)

Une machine de Turing à $k \geq 2$ rubans est un octuplet $M = (\Sigma, \Gamma, B, Q, q_0, q_a, q_r, \delta)$ où :

- Σ est un ensemble fini non vide appelé *alphabet d'entrée* (ce sont les symboles utilisés pour écrire le mot d'entrée) ;
- Γ est un ensemble fini appelé *alphabet de travail*, tel que $\Sigma \subset \Gamma$ (ce sont les symboles utilisés dans les cases au cours du calcul) ;
- B est un symbole spécial *blanc* (représentant une case vide) tel que $B \in \Gamma \setminus \Sigma$;
- Q est un ensemble fini appelé *ensemble des états* (les états que peuvent prendre les têtes de lecture/écriture) ;

- $q_0 \in Q$ est un état spécial appelé *état initial* (indiquant l'état dans lequel les têtes commencent le calcul) ;
- $q_a \in Q$ et $q_r \in Q$ sont des états spéciaux appelé *états terminaux* (indiquant la fin du calcul) : q_a est l'état *d'acceptation* et q_r l'état de *rejet* ;
- $\delta : (Q \setminus \{q_a, q_r\}) \times \Gamma^{k-1} \rightarrow Q \times \Gamma^{k-1} \times \{G, S, D\}^k$ est la *fonction de transition* décrivant le comportement des têtes :

$$\delta(q, a_1, \dots, a_{k-1}) = (r, b_2, \dots, b_k, d_1, \dots, d_k)$$

si, lorsque les têtes sont dans l'état q et que la i -ème tête lit le symbole a_i ($i \leq k-1$ car la tête du ruban de sortie ne lit rien), alors le nouvel état des têtes est r , la tête i écrit le symbole b_i dans la case à la place de a_i ($i \geq 2$ car la tête du ruban d'entrée n'écrit rien) et elle se déplace dans la direction d_i (G à gauche, S sur place, D à droite).

1-D Remarques

- La fonction de transition δ n'est pas définie sur les états q_a et q_r puisque ce sont les états terminaux qui marquent la fin du calcul : la machine s'arrête dès que l'un d'eux est atteint.
- De plus, il est commode d'utiliser un alphabet de travail plus étoffé que celui d'entrée afin de disposer de symboles permettant de séparer des mots, de marquer des positions, etc.
- Enfin, insistons une dernière fois sur le fait que, dans le modèle que nous avons choisi, le ruban d'entrée est en lecture seule (donc la fonction de transition ne spécifie aucun symbole à écrire sur le ruban d'entrée) et le ruban de sortie en écriture seule (donc la fonction de transition ne spécifie aucun symbole à lire sur le ruban de sortie).

Les rubans de M sont infinis à droite et à gauche :

- sur chaque ruban, une case est repérée par un entier relatif $i \in \mathbb{Z}$. La *position d'une tête* à un instant donné est le numéro de la case sur laquelle elle est située. Toutes les têtes sont en position 1 au début du calcul.

À chaque transition, la machine évolue d'une configuration à une autre. Pour décrire la configuration de la machine, il faut décrire le contenu des rubans, la position des têtes et leur état :

- une *configuration* C de M est un élément de $Q \times (\Gamma^k)^{\mathbb{Z}} \times \mathbb{Z}^k$: C décrit l'état des têtes, le contenu de chaque case des k rubans et les positions des têtes.