

Machine de Turing



Nous avons vu qu'un programme peut être considéré comme la **décomposition** de la tâche à réaliser en une séquence d'instructions **élémentaires** (manipulant des données élémentaires) **compréhensibles** par l'automate programmable que l'on désire utiliser.

Cependant, chaque automate programmable (en pratique chaque processeur) se caractérise par un jeu d'instructions élémentaires qui lui est propre.

Dans la pratique, cette diversité est résolue par l'existence de compilateurs ou d'interpréteurs qui traduisent les instructions du langage (évolué) mis à disposition des programmeurs en les instructions élémentaires (langage machine) du processeur utilisé.

Cependant, cette solution n'est pas suffisante pour les besoins de l'informatique théorique, qui requière une représentation unifiée de la notion d'automate programmable, permettant de démontrer des résultats généraux (décidabilité, complexité, ...) vrais pour l'ensemble des automates programmables concrets que l'on peut envisager.

A cette fin a été développée la notion de *machine de Turing*, qui constitue une abstraction (et une formalisation) de la notion d'automate programmable.



Machine de Turing: définition

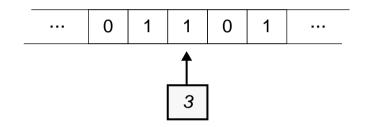


Une machine de Turing est un automate abstrait, constitué des éléments suivants:

- une bande infinie, décomposée en cellules au sein desquelles peuvent être stockés des caractères (issus d'un ensemble fini).
- une tête de lecture/écriture, pouvant:
 - lire et modifier le caractère stocké dans la cellule correspondant à la position courante de la tête (le *caractère courant*)
 - se déplacer d'une cellule vers la gauche ou vers la droite (modifier la position courante)
- un ensemble fini d'états internes permettant de conditionner le fonctionnement de la machine
- une table de transitions indiquant, pour chaque couple (état interne, caractère courant) les nouvelles valeurs pour ce couple, ainsi que le déplacement de la tête de lecture/écriture.

Dans la table de transisitions, chaque couple est donc associé à un triplet:

(état interne[nouveau], caractère[nouveau], déplacement)



caractère état courant courant	0	1	3
1	(1,0,+)	(1,0,+)	(2,ε,-)
2	(2,0,-)	(2,0,-)	(3,£,+)

avec + respect. - indiquant un déplacement vers la droite, respect. vers la gauche.



Machine de Turing: fonctionnement



Une machine de Turing fonctionne selon le principe suivant:

- 1) la bande est initialisée avec la séquence de caractères correspondant aux données d'entrées;
- 2) la tête de lecture/écriture est positionnées sur la première cellule de la bande, et l'état interne est positionné à sa valeur initiale (par exemple 1)
- 3) tant que le couple courant (état interne, caractère courant) est présent dans la table de transitions, le triplet (état interne_[nouveau], caractère_[nouveau], déplacement) est utilisé pour mettre à jour l'état interne, le caractère courant, puis le déplacement de la tête de lecture/écriture est effectué.
- 4) lorsqu'est rencontré un couple (état interne, caractère courant) non recensé dans la table de transitions, la machine s'arrète et la séquence de caractères stockée à ce moment-là sur la bande est considérée comme le résultat du traitement.



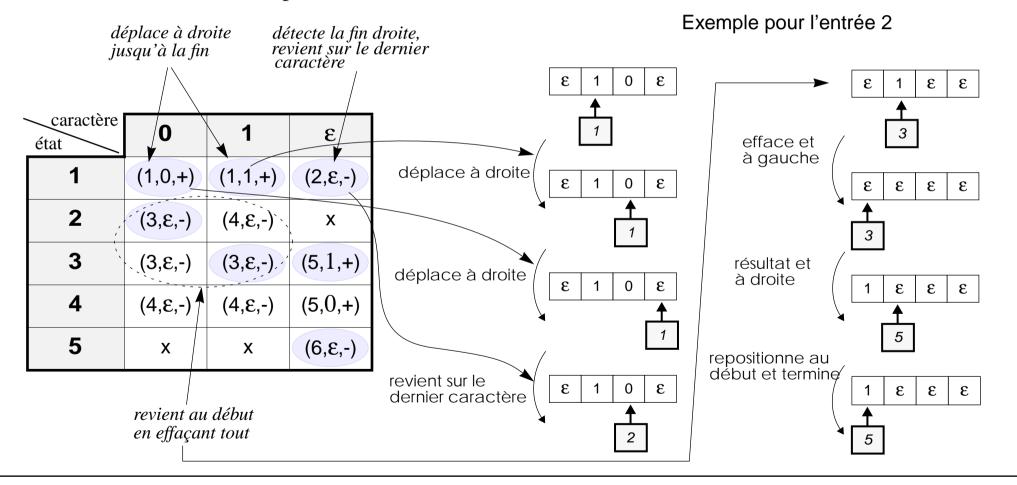
Machine de Turing: exemple



Exemple: une machine de Turing déterminant si un nombre est pair

Entrée: le nombre à tester, sous forme binaire

Sortie: 1 si le nombre est pair, 0 sinon





Machine de Turing: forme canonique



On voit qu'une machine de Turing est caractérisée par

- 1. sa logique de fonctionnement;
- 2. le codage de ses entrées et sorties, sous forme de séquences de caractères;
- 3. la table de transitions décrivant son fonctionnement.

Si l'on impose de coder les entrées et les sorties sous forme binaire et d'indiquer l'absence de caractère dans une cellule par le caractère ε , on obtient une représentation uniforme des machines de Turing, appelée représentation canonique.

D'autres choix sont possibles pour la définition des machines de Turing (plusieurs bandes, autres opérations élémentaires, autres alphabets de caractères, ...) mais on peut montrer que les résultats théoriques obtenus avec de telles machines sont équivalents à ceux obtenus à l'aide d'une machine de Turing canonique.



Machine de Turing universelle (1)



Le fonctionnement d'une machine de Turing est conditionné par sa table de transition.

une machine de Turing constitue donc une abstraction pour la notion **d'automate** *modifiable*...

... mais pas encore pour celle d'automate programmable, pour laquelle le programme doit faire partie des données d'entrée de la machine, et non pas représenter un élément constitutif de l'automate (comme c'est le cas pour la table de transitions d'une machine de Turing)

Si l'on désire qu'une machine de Turing constitue une abstraction pour la notion d'automate **programmable**, il faut donc que sa table de transitions soit fixe, et que le conditionnement de son fonctionnement soit **entièrement** imposé par ses données d'entrées.

mais comment construire une telle machine, alors appelée *machine de Turing universelle* ?



Machine de Turing universelle (2)



On peut montrer que l'on peut effectivement construire une *machine de Turing universelle* permettant de simuler le fonctionnement d'une machine de Turing quelconque, i.e. d'avoir le même comportement, pour ce qui est des entrées et des sorties, tout en ayant une table de transitions fixe.

L'idée permettant de construire une telle machine T_o est la suivante:

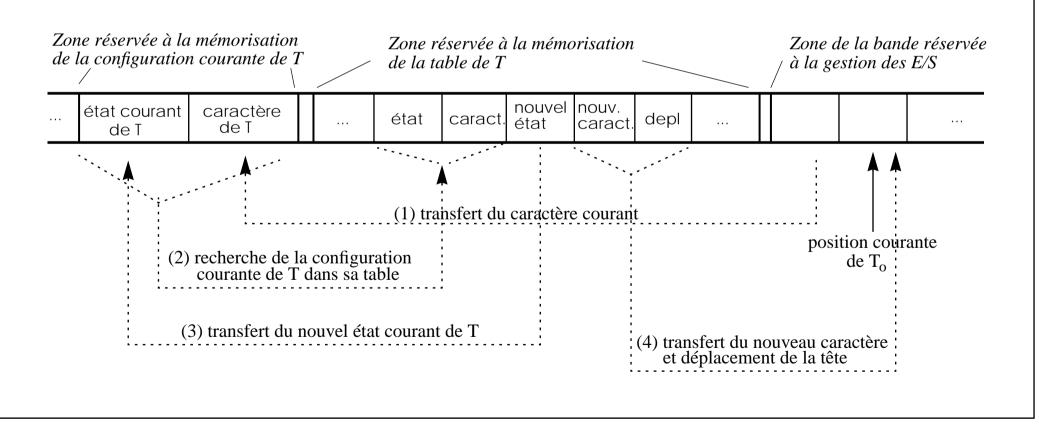
- 1. la table de transitions de la machine T à simuler est codée sous la forme d'une séquence binaire;
- 2. la bande de la machine universelle est séparée en deux zones distinctes, l'une permettant de stocker la séquence binaire représentant la table de la machine T, ainsi que sa configuration (état interne, caractère courant), et l'autre permettant de gérer les véritables entrées/sorties.



Machine de Turing universelle (3)



Le fonctionnement de la machine universelle peut alors être schématisé ainsi:





Machine de Turing universelle (4)



La notion de machine de Turing universelle T_o permet d'introduire les notations utiles suivantes:

Soit T une machine de Turing, on noterai (T) la valeur entière correspondant à la séquence binaire codant la table de transitions de T

Si i est une valeur entière correspondant au codage d'une table de transitions, on notera T_i la machine de Turing universelle correspondante.

On a donc: $T = T_{i(T)}$

Si i est une valeur entière correspondant au codage d'une séquence d'entrée pour une machine de Turing T, on notera T(i) la séquence de sortie produite par l'application de T à la séquence d'entrée i (attention: T(i) peut ne pas être définie, si T ne termine pas sur i)

Si T_o est la machine de Turing universelle, T une machine de Turing et j une séquence d'entrée pour T_o , on notera $T_o(i(T),j)$ le résultat de la **simulation** de T sur j par T_o On a donc: $T_o(i(T),j) = T(j)$

(lorsque ces valeurs sont définies)



Intérêt des machines de Turing



Les machines de Turing constituent une notion centrale en informatique, car:

elles permettent de donner une base théorique solide aux notions importantes de *décidabilité* et de *complexité* (notions évoquées plus en détails plus loin dans ce cours);

elles permettent de donner un contenu précis à la notion informelle d'algorithme



Algorithme: définition (1)



Algorithme: (définition informelle)

un algorithme, pour une tâche donnée, est la décomposition de cette tâche en une séquence finies d'étapes élémentaires permettant de la réaliser.

Si la tâche peut être **formalisée** sous la forme d'une association entre des entrées et des sorties, alors produire une machine de Turing permettant de réaliser cette association peut être vu comme une façon possible de résoudre la tâche.

La **table de transitions** de la machine produite est donc une **représentation particulière possible de l'algorithme** de résolution associé.



Algorithme: définition (2)



Les machines de Turing fournissent donc, par définition, une représentation possible pour les algorithmes de résolution des tâches pouvant être traitées par des machines de Turing.

De plus, l'expérience acquise dans le domaine de l'informatique invite à penser que les machines de Turing constituent en fait un **outil encore plus puissant**:

Thèse de Church: tout algorithme peut être représenté

sous la forme d'une table de transitions

d'une machine de Turing.

L'idée sous-jacente est que les machines de Turing, malgré leur apparente simplicité, possèdent en fait une puissance suffisante pour permettre de résoudre tout problème pouvant être traité de façon automatique.

une formulation savante de la thèse de Church est doc:

«tout problème calculable est Turing calculable »



Algorithmique



Un *algorithme* est:

une **suite finie** de règles à appliquer, dans un **ordre déterminé**, à un **nombre fini** de données,

pour arriver, en un nombre fini d'étapes, à un résultat, et cela quelque soit les données traitées. [Encyclopedia Universalis]

Un algorithme correspond donc à la partie **conceptuelle** d'un programme, indépendante du langage de programmation.

Le programme est alors **l'implémentation** de l'algorithme, dans un langage de programmation (et sur un système) particulier.



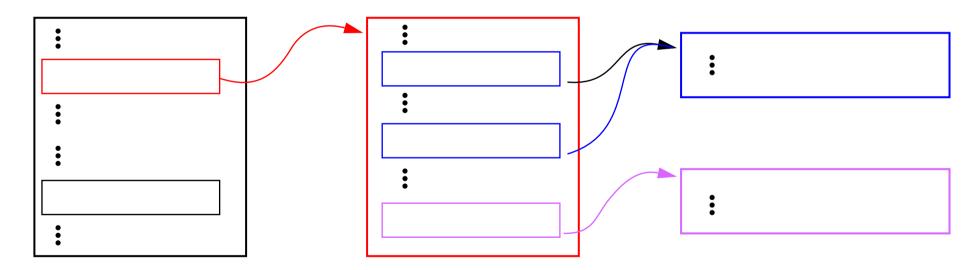
Approche descendante



Méthode de résolution: (élaboration de solutions algorithmiques)

On résoud généralement un problème par une *analyse descendante*, c'est-à-dire une analyse du plus général au plus spécifique.

Cette analyse se fait à l'aide de **blocs imbriqués**, l correspondant à des traitements de plus en plus spécifiques, eux aussi décrits à l'aide **d'algorithmes**. 2



^{1.} Notons le parallèle évident avec l'implémentation du programme décomposé en fonctions

^{2.} La méthode de résolution est donc itérative: on résoud par analyses successives, à des niveaux de détails de plus en plus précis.



Exemple: tri par insertion

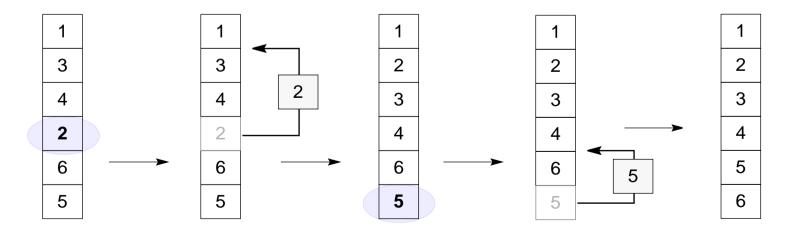


Un algorithme: tri (d'un tableau) par insertion

le principe du tri par insertion est extrêmement simple:

«tout élément mal placé dans le tableau va être déplacé vers (inséré à) sa bonne place»

- Par «élément mal placé», on entend ici un élément pour lequel la relation d'ordre (utilisée pour le tri) avec l'élément prédécesseur n'est pas satisfaite... par exemple, un élément strictement plus petit que son prédécesseur, pour un tri suivant l'ordre croissant.
- Par «sa bonne place», on entend ici une position –parmis les éléments déjà triés– pour laquelle la relation d'ordre entre le prédécesseur et l'élément est satisfaite, ainsi qu'entre l'élément et son successeur.





tri par insertion: résolution globale



Le schéma général de l'algorithme de tri par insertion est le suivant:

tri insertion

entrée: un tableau d'entiers quelconque

sortie: un tableau d'entiers trié (mêmes éléments que l'entrée)

chercher un élément *mal placé*

tant qu'un tel élément existe:

chercher sa bonne place

déplacer l'élément vers sa bonne place

chercher un élément *mal placé*

fin tant que



tri par insertion: résolution détaillée (1)



• Le bloc *mal placé* prend en entrée un tableau tab, et donne en sortie l'indice du premier élément de tab de prédécesseur strictement plus grand. Par convention, si'il n'existe pas d'élément admettant un tel sucesseur, la sortie sera MAX, la taille du tableau.

• Comme le 1^{er} élément de tab ne peut être mal placé (puisque sans prédécesseur), le bloc mal placé doit donc parcourir les éléments de tab, à la recherche d'un élément mal placé, en partant du second élément.

• Le bloc *mal placé* correspond donc à une itération sur les éléments de tab, du deuxième élément au dernier (d'indice MAX-1).



tris par insertion: résolution détaillée (2)



Un algorithme pour le bloc *mal placé* sera:

mal placé

entrée: un tableau d'entiers

sortie: l'indice du premier élément mal placé

pour chaque élément du tableau, à partir du deuxième:

comparer cet élément au précédent

s'il est plus petit, alors il est mal placé:

retourner l'indice de cet élément (et terminer)

fin si

fin pour

retourner MAX, la taille du tableau (et terminer)



tri par insertion: résolution détaillée (3)



• Le bloc bonne place prend en entrée un tableau tab et l'indice pos d'un élément mal placé, et donne en sortie l'indice newpos de la «bonne place» de l'élement dans le tableau.

• La «bonne place» correspond à la position d'indice newpos (<pos) le plus grand tel que tab[newpos-1] <= tab[pos], où 0 si cette relation n'est pas satisfaite pour newpos = 1.

• Le bloc bonne place doit donc parcourir les positions de tab, entre le premier élément et (au maximum) pos, à la recherche de la «bonne position»

• Le bloc bonne place correspond donc à une itération sur les éléments de tab, du premier élément à celui d'indice pos.



tri par insertion: résolution détaillée (4)



Un algorithme pour le bloc bonne place sera:³

bonne place entrée: un tableau d'entiers tab, l'indice pos d'un élément mal placé sortie: l'indice de la bonne place de l'élément pour chaque élément du tableau, de 0 à pos: si l'élément est plus grand que tab[pos]: retourner l'indice de cet élément (et terminer) fin si fin pour

^{3.} L'algorithme correspond à une interprétation (équivalente en terme de résultat) de la description du transparent précédent !

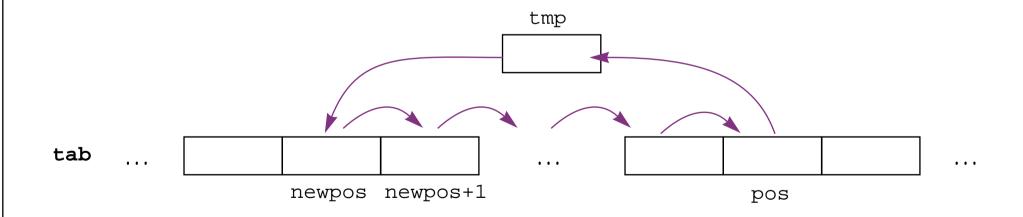


tri par insertion: résolution détaillée (5)



• Le bloc déplacer prend en entrée un tableau tab, l'indice pos d'origine d'un élément, et l'indice newpos de destination de cet élément, et doit déplacer ledit élément de sa position d'origine à sa positition de destination.

On peut effectuer cette opération par **décalages successifs** (en utilisant un stockage temporaire tmp)





tri par insertion: résolution détaillée (6)



Un algorithme pour le bloc | déplacer | sera

déplacer

entrée: un tableau, un indice d'origine pos, un indice final newpos

sortie: le tableau dans lequel l'élement d'indice d'origine à été déplacé et

possède maintenant l'indice final (l'ordre des autres éléments

est conservé)

mémoriser l'élément d'indice pos

pour chaque élément du tableau entre pos-1 et newpos

copier cet élément à la position d'indice courant+1

fin pour

copier l'élément mémorisé à la position d'indice newpos



Paradigme «diviser pour résoudre» (1)



Il n'existe pas de méthode miracle pour fabriquer les solutions algorithmiques requises pour les problèmes auxquels l'informaticien peut être confronté.

Cependant, un certain nombre de schémas généraux de résolution peuvent être utilisés.

Parmi ces schémas, on peut citer le paradigme «diviser pour résoudre» (divide and conquer), qui est fréquemment utilisé.



Diviser pour résoudre (2)



Le schéma général d'une approche «diviser pour résoudre» de résolution d'un problème P appliqué à des données X est le suivant:

P(X):

si X est suffisement simple ou petit, appliquer adhoc(X), sinon:

décomposer X en instances plus petites, $x_1, ..., x_n$ puis pour tout x_i (1 <= i <= n): résoudre $P(x_i)$ pour obtenir une solution partielle y_i recombiner les y_i pour obtenir une solution globale Y pour P(X)

adhoc(X)

est un algorithme permettant de résoudre P pour des données X simples.



Diviser pour résoudre: exemple (1)



Un premier exemple d'application du paradigme «diviser pour résoudre» à déjà été présenté, à l'occasion de l'étude des fonctions récursives, avec le tri récursif par insertion:

tri(X):

si X est vide ou réduit à 1 seul élément, X est solution sinon:

décomposer X en:

 $x_1 = \{X[1]\}$ où X[1] est le 1^{er} élement de X,

 $\mathbf{x}_2 = \mathbf{X} \setminus \mathbf{X}[1]$

résoudre $tri(x_1)$ (cas trivial) -> y_1

résoudre $tri(x_2)$ (appel récursif) -> y_2

produire une solution pour X en insérant y₁ à sa bonne place dans y₂, et retourner la combinaison obtenue.



Diviser pour résoudre: exemple (2)



Recherche par dichotomie:

Objectif: déterminer si un élément x appartient à un ensemble E d'éléments $E = \{x_1, ..., x_n\}$

Entrée: x et E (sous la forme d'un tableau trié)

Sortie: vrai ou faux selon que x appartient ou non à E.

Schéma de résolution:

```
\begin{array}{l} \textit{appartient}(x,E) \colon\\ \textbf{si } E \text{ est vide,} \\ & \text{ la solution est } \textit{faux} \\ \textbf{sinon } \textbf{si } E = \{x_1\}, \\ & \text{ la solution est vrai si } \textit{x} = \textit{x}_1, \text{ faux sinon} \\ \textbf{sinon } (E = \{x_1, ..., x_n\}) \\ & \text{ décomposer } E \text{ en:} \\ & E_1 = \{x_{n/2}\} \text{ avec } n/2 \text{ la division entière de n par } 2 \text{ !} \\ & E_2 = \{x_1, ..., x_{(n/2)-1}\} \\ & E_3 = \{x_{(n/2)+1}, ..., x_n\} \\ & \textbf{si } y_1 = \textit{appartient}(x, E_1) \text{ est } \textit{vrai}, \text{ alors } Y \text{ est } \textit{vrai} \\ & \textbf{sinon si } \textit{x} < x_{n/2} \text{ alors } Y = y_2 = \textit{appartient}(\textit{x}, E_2) \\ & \textbf{sinon } Y = y_3 = \textit{appartient}(\textit{x}, E_3) \\ & \text{ (la solution est } Y) \end{array}
```