ALGORITHMIQUE - Partie 3
Les Tests

Rappel : il n'existe que 4 briques :

L'AFFECTATION DE VARIABLES
LA LECTURE / ECRITURE
LES TESTS
LES BOUCLES

Nous pouvons donner à notre ordinateur des séries d'instructions à effectuer selon que la situation se présente d'une manière ou d'une autre. Cette structure logique s'appelle un **test** ou **structure alternative**.

2. Structure d'un test

Il n'y a que **deux formes possibles** pour un test

Si booléen Alors
Instructions
Finsi

Si booléen Alors
Instructions 1
Sinon
Instructions 2
Finsi

Un **booléen** est une **expression** dont la valeur est VRAI ou FAUX. Cela peut donc être :

- une **variable** (ou une expression) de type booléen
- une **condition**

Nous reviendrons sur ce qu'est une **condition** en informatique.

A la première ligne (Si… Alors) la machine examine la valeur du booléen. Si ce booléen a pour valeur VRAI, elle exécute la série d'instructions. Cette série d'instructions peut être très brève comme très longue. En revanche, dans le cas où le booléen est faux, l'ordinateur saute directement aux instructions situées après le FinSi.

Dans le cas de la structure avec *sinon*, c'est à peine plus compliqué. Dans le cas où le booléen est VRAI, et après avoir exécuté la série d'instructions 1, au moment où elle arrive au mot « Sinon », la machine saute directement à la première instruction située après le « Finsi ». De même, au cas où le booléen a comme valeur « Faux », la machine saute directement à la première ligne située après le « Sinon » et exécute l'ensemble des « instructions 2 ». Dans tous les cas, les instructions situées juste après le FinSi seront exécutées normalement.

Qu'est ce qu'une condition ?

Une condition est une comparaison

Cette définition est essentielle ! Elle signifie qu'une condition est composée de trois éléments :

- une valeur
- un **opérateur de comparaison**
- une autre valeur

Les valeurs peuvent être a priori de n'importe quel type (numériques, caractères…). Mais si l'on veut que la comparaison ait un sens, il faut que les deux valeurs de la comparaison soient du même type !

Les **opérateurs de comparaison** sont :

égal à…

différent de…

strictement plus petit que…

strictement plus grand que…

plus petit ou égal à…

plus grand ou égal à…

L'ensemble des trois éléments composant la condition constitue donc, si l'on veut, une affirmation, qui à un moment donné est VRAIE ou FAUSSE.

À noter que ces opérateurs de comparaison peuvent tout à fait s'employer avec des caractères. Ceux-ci sont codés par la machine dans l'ordre alphabétique (code ASCII), les majuscules étant systématiquement placées avant les minuscules. Ainsi on a :

"t" < "w" VRAI
"Maman" > "Papa" FAUX
"maman" > "Papa" VRAI

Remarque
En formulant une condition dans un algorithme, il faut se méfier de certains raccourcis du langage courant, ou de certaines notations valides en mathématiques, mais qui mènent à des non-sens informatiques. Prenons par exemple la phrase « Toto est compris entre 5 et 8 ». On peut être tenté de la traduire par : ***5 < Toto < 8***
Or, une telle expression, qui a du sens en français, comme en mathématiques, ***ne veut rien dire en programmation****. En effet, elle comprend deux opérateurs de comparaison, soit un de trop, et trois valeurs, soit là aussi une de trop. On va voir comment traduire convenablement une telle condition.*

Exercice 3.1

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on laisse de côté le cas où le nombre vaut zéro).

Conditions composées – utilisation des opérateurs logiques

Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme simple exposée ci-dessus. Reprenons le cas « **Toto est inclus entre 5 et 8** ». En fait cette phrase cache non une, mais deux conditions. Car elle revient à dire que « Toto est supérieur à 5 et Toto est inférieur à 8 ». Il y a donc bien là deux conditions, reliées par ce qu'on appelle un **opérateur logique**, le mot **ET**.

Comme on l'a évoqué, l'informatique met à notre disposition quatre opérateurs logiques : **ET, OU, NON, et XOR**.

Le **ET** a le même sens en informatique que dans le langage courant. Pour que "Condition1 ET Condition2" soit VRAI, il faut impérativement que Condition1 soit VRAI et que Condition2 soit VRAI.

Le **OU** :. Pour que "Condition1 OU Condition2" soit VRAI, il suffit que Condition1 soit VRAIE ou que Condition2 soit VRAIE. Le point important est que si Condition1 est VRAIE et que Condition2 est VRAIE aussi, Condition1 OU Condition2 reste VRAIE.

Le **XOR** (ou OU exclusif) fonctionne de la manière suivante. Pour que "Condition1 XOR Condition2" soit VRAI, il faut que soit Condition1 soit VRAI, soit que Condition2 soit VRAI. Si toutes les deux sont fausses, ou que toutes les deux sont VRAI, alors le résultat global est considéré comme FAUX.

Le **NON** inverse une condition : NON(Condition1)est VRAI si Condition1 est FAUX, et il sera FAUX si Condition1 est VRAI. C'est l'équivalent pour les booléens du signe "moins" que l'on place devant les nombres.

On représente fréquemment tout ceci dans des **tables de vérité** (C1 et C2 représentent deux conditions, et on envisage à chaque fois les quatre cas possibles)

C1 et C2	C1	C2
Faux	Faux	Faux
Faux	Faux	Vrai
Faux	Vrai	Faux
Vrai	Vrai	Vrai

C1 ou C2	C1	C2
Faux	Faux	Faux
Vrai	Faux	Vrai
Vrai	Vrai	Faux
Vrai	Vrai	Vrai

C1 xor C2	C1	C2
Faux	Faux	Faux
Vrai	Faux	Vrai
Vrai	Vrai	Faux
Faux	Vrai	Vrai

Non C1	C1
Vrai	Faux
Faux	Vrai

Exercice 3.2

Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si leur produit est négatif ou positif (on laisse de côté le cas où le produit est nul). Attention toutefois : on ne doit **pas** calculer le produit des deux nombres.

Exercice 3.3

Ecrire un algorithme qui demande trois noms à l'utilisateur et l'informe ensuite s'ils sont rangés ou non dans l'ordre alphabétique.

Tests imbriqués

Graphiquement, on peut très facilement représenter un SI comme un aiguillage de chemin de. Un SI ouvre donc deux voies, correspondant à deux traitements différents. Mais il y a des tas de situations où deux voies ne suffisent pas. Par exemple, un programme devant donner l'état de l'eau selon sa température doit pouvoir choisir entre trois réponses possibles (solide, liquide ou gazeuse).
Une première solution serait la suivante :

```
Variable Temp en Entier
Début
Ecrire "Entrez la température de l'eau :"
Lire Temp
Si Temp =< 0 Alors
  Ecrire "C'est de la glace"
FinSi
Si Temp > 0 Et Temp < 100 Alors
  Ecrire "C'est du liquide"
Finsi
Si Temp > 100 Alors
  Ecrire "C'est de la vapeur"
Finsi
Fin
```

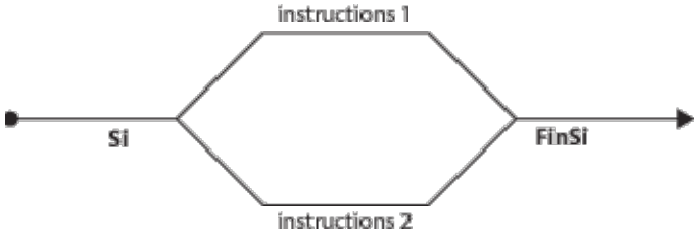
C'est un peu laborieux. Les conditions se ressemblent plus ou moins, et surtout on oblige la machine à examiner trois tests successifs alors que tous portent sur une même chose, la température de l'eau (la valeur de la variable Temp).
Il serait ainsi bien plus rationnel d'imbriquer les tests de cette manière :

```
Variable Temp en Entier
Début
Ecrire "Entrez la température de l'eau :"
Lire Temp
Si Temp =< 0 Alors
   Ecrire "C'est de la glace"
Sinon
   Si Temp < 100 Alors
      Ecrire "C'est du liquide"
   Sinon
      Ecrire "C'est de la vapeur"
   Finsi
Finsi
Fin
```

au lieu de devoir tester trois conditions, dont une composée, nous n'avons plus que deux conditions simples. Mais aussi, et surtout, nous avons fait des économies sur le temps d'exécution de l'ordinateur. Si la température est inférieure à zéro, celui-ci écrit dorénavant « C'est de la glace » et passe directement à la fin, sans être ralenti par l'examen d'autres possibilités (qui sont forcément fausses).
Cette deuxième version n'est donc pas seulement plus simple à écrire et plus lisible, elle est également plus performante à l'exécution.
Les structures de tests imbriqués sont donc un outil indispensable à la simplification et à l'optimisation des algorithmes.

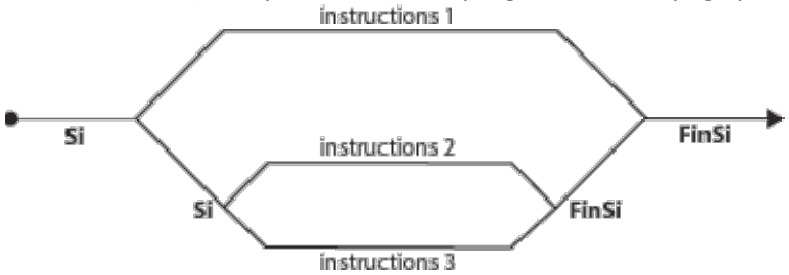
SINONSI

Une structure SI peut se schématiser ainsi :



Mais dans certains cas, ce ne sont pas deux voies qu'il nous faut, mais trois, ou même plus. Dans le cas de l'état de l'eau, il nous faut trois voies pour notre « train », puisque l'eau peut être solide, liquide ou gazeuse. Alors, nous n'avons pas eu le choix : pour deux voies, il nous fallait un aiguillage, pour trois voies il nous en faut deux, imbriqués l'un dans l'autre.

Cette structure (telle que nous l'avons programmée à la page précédente) devrait être schématisée comme suit :



Soyons bien clairs : cette structure est la seule possible du point de vue. Mais du point de vue de l'écriture, le pseudo-code algorithmique admet une simplification supplémentaire. Ainsi, il est possible (mais non obligatoire, que l'algorithme initial :

```
Si Temp =< 0 Alors
  Ecrire "C'est de la glace"
Sinon
  Si Temp < 100 Alors
    Ecrire "C'est du liquide"
  Sinon
    Ecrire "C'est de la vapeur"
  Finsi
Finsi
```

devienne :

```
Si Temp =< 0 Alors
  Ecrire "C'est de la glace"
SinonSi Temp < 100 Alors
  Ecrire "C'est du liquide"
Sinon
  Ecrire "C'est de la vapeur"
Finsi
```

le cas de tests imbriqués, le Sinon et le Si peuvent être fusionnés en un SinonSi. On considère alors qu'il s'agit d'un seul bloc de test, conclu par un seul FinSi

Le SinonSi permet en quelque sorte de créer (en réalité, de simuler) des aiguillages à plus de deux branches. On peut ainsi enchaîner les SinonSi les uns derrière les autres pour simuler un aiguillage à autant de branches que l'on souhaite.

Variables Booléennes

Jusqu'ici, pour écrire nos tests, nous avons utilisé uniquement des conditions. Mais vous vous rappelez qu'il existe un type de variables (les booléennes) susceptibles de stocker les valeurs VRAI ou FAUX. En fait, on peut donc entrer des conditions dans ces variables, et tester ensuite la valeur de ces variables.

Reprenons l'exemple de l'eau. On pourrait le réécrire ainsi :

```
Variable Temp en Entier
Variables A, B en Booléen
Début
Ecrire "Entrez la température de l'eau :"
Lire Temp
A ← Temp =< 0
B ← Temp < 100
Si A Alors
  Ecrire "C'est de la glace"
SinonSi B Alors
  Ecrire "C'est du liquide"
Sinon
  Ecrire "C'est de la vapeur"
Finsi
Fin
```

A priori, cette technique ne présente guère d'intérêt : on a alourdi plutôt qu'allégé l'algorithme de départ, en ayant recours à deux variables supplémentaires.
Mais souvenons-nous : une variable booléenne n'a besoin que d'un seul bit pour être stockée. De ce point de vue, l'alourdissement n'est donc pas considérable.
Dans certains cas, notamment celui de conditions composées très lourdes (avec plein de ET et de OU tout partout) cette technique peut faciliter le travail du programmeur, en améliorant nettement la lisibilité de l'algorithme.

Exercices :

Exercice 3.4

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on inclut cette fois le traitement du cas où le nombre vaut zéro).

Exercice 3.5

Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si le produit est négatif ou positif (on inclut cette fois le traitement du cas où le produit peut être nul). Attention toutefois, on ne doit pas calculer le produit !

Exercice 3.6

Ecrire un algorithme qui demande l'âge d'un enfant à l'utilisateur. Ensuite, il l'informe de sa catégorie :
"Poussin" de 6 à 7 ans
"Pupille" de 8 à 9 ans
"Minime" de 10 à 11 ans
"Cadet" après 12 ans