



Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Rue Royale, 67. 1000 Bruxelles
02/219.15.46 – esi@he2b.be

Algorithmique

2019

Bachelor en Informatique
DEV2

M. Codutti (MCD), H. Delannoy (HDE),
S. Drobisz (SDR), A. Paquot (APA) & N. Richard (NRI)

Table des matières

1	Les tableaux à 2 dimensions	3
1.1	Définition	3
1.2	Notations	3
1.3	La troisième dimension (et au-delà)	6
1.4	Parcours d'un tableau à deux dimensions	6
1.5	Exercices	11
2	L'orienté objet	15
2.1	Motivation	15
2.2	Illustration : une durée	15
2.3	Quelques éléments de syntaxe	19
2.4	Mise en pratique : le lièvre et la tortue	20
3	La liste	27
3.1	La classe List	28
3.2	Exercices	30
4	Les traitements de rupture	33
4.1	Le classement complexe	33
4.2	La notion de rupture	35
4.3	Traitement des ruptures dans une séquence ordonnée	35
4.4	Traitements de clôture et d'initialisation	37
4.5	Exercices	37
5	Représentation des données	41
5.1	Se poser les bonnes questions	41
5.2	Les structures de données	42
5.3	Exercices	42
A	Compléments	51
A.1	Énumération	51
A.2	Gestion des erreurs	52
B	Aide mémoire	53

Les tableaux à 2 dimensions


1.1 Définition

La **dimension** d'un tableau est le nombre d'indices qu'on utilise pour faire référence à un de ses éléments. Attention de ne pas confondre avec la taille ! 

En DEV1, nous avons introduit les tableaux à une dimension. Un seul indice suffisait à localiser un de ses éléments. Pour le dire autrement, chaque case possédait **un** numéro. De nombreuses situations nécessitent cependant l'usage de tableaux à deux dimensions. Ils vous sont déjà familiers par leur présence dans beaucoup de situations courantes : calendrier, grille horaire, grille de mots croisés, sudoku, jeux se déroulant sur un quadrillage (damier, échiquier, scrabble...). Dans ces situations, chaque case est désignée par **deux** numéros.

1.2 Notations

1.2.1 Déclarer

Pour **déclarer** un tableau à 2 dimensions, on écrira : 

```
nomTableau: array of nbLignes × nbColonnes TypeElément
```


où nbLignes et nbColonnes sont des expressions entières quelconques.

Exemple :

```
tab: array of 5×10 integers
```

déclare un tableau de 5 lignes par 10 colonnes dont chaque case contient un entier.

1.2.2 Utiliser

Pour **accéder** à une case du tableau on donnera les deux indices entre crochets. Comme en DEV1, on considère que la première ligne et la première colonne portent le numéro (l'indice) 0. 

Exemple :

```
print tab[2,4] // affiche le 5° élément de la 3° ligne du tableau nommé tab.
```

1.2.3 Visualiser

Notez que la vue sous forme de tableau avec des lignes et des colonnes est une vision humaine. Il n'y a pas de lignes ni de colonnes en mémoire. Pour être précis, on devrait juste parler de première dimension et de deuxième dimension mais la notion de ligne et de colonne est un abus de langage qui simplifie le discours.

On pourrait aussi visualiser un tableau à deux dimensions comme un tableau à une dimension dont chacun des éléments est lui-même un tableau à une dimension.

Exemple : Soit le tableau déclaré ainsi :

nombres: **array of 4×5 integers**

On peut le visualiser à l'aide d'une grille à 4 lignes et 5 colonnes.

	0	1	2	3	4
0	0	1	2	3	4
1	10	11	12	13	14
2	20	21	22	23	24
3	30	31	32	33	34

Ainsi, la valeur de `nombres[2,3]` est la valeur 23.

La vision « tableau de tableaux » (ou décomposition en niveaux) donnerait :

0	1	2	3
0 1 2 3 4 0 1 2 3 4	0 1 2 3 4 10 11 12 13 14	0 1 2 3 4 20 21 22 23 24	0 1 2 3 4 30 31 32 33 34

Dans cette représentation, le tableau `nombres` est d'abord décomposé à un premier niveau en quatre éléments auxquels on accède par le premier indice. Ensuite, chaque élément de premier niveau est décomposé en cinq éléments de deuxième niveau accessibles par le deuxième indice.

1.2.4 Exemples

Exemple 1 – Remplir les coins. Dans ce petit exemple, on a un tableau de chaînes et on donne des valeurs aux coins.

"NO"				"NE"
"SO"				"SE"

```
// Déclare un tableau et donne des valeurs aux coins.
```

```
algorithm remplirCoins
```

```
  grille: array of 3×5 string
```

```
  grille[0,0] = "NO"
```

```
  grille[0,4] = "NE"
```

```
  grille[2,0] = "SO"
```

```
  grille[2,4] = "SE"
```

```
end
```

Exemple 2 – Gestion des stocks. Reprenons l'exemple du stock de 10 produits qui a servi d'introduction au chapitre sur les tableaux mais, cette fois, pour chaque jour de la semaine.

	article0	article1	article2	...	article7	article8	article9
lundi	cpt[0,0]	cpt[0,1]	cpt[0,2]	...	cpt[0,7]	cpt[0,8]	cpt[0,9]
mardi	cpt[1,0]	cpt[1,1]	cpt[1,2]	...	cpt[1,7]	cpt[1,8]	cpt[1,9]
mercredi	cpt[2,0]	cpt[2,1]	cpt[2,2]	...	cpt[2,7]	cpt[2,8]	cpt[2,9]
jeudi	cpt[3,0]	cpt[3,1]	cpt[3,2]	...	cpt[3,7]	cpt[3,8]	cpt[3,9]
vendredi	cpt[4,0]	cpt[4,1]	cpt[4,2]	...	cpt[4,7]	cpt[4,8]	cpt[4,9]
samedi	cpt[5,0]	cpt[5,1]	cpt[5,2]	...	cpt[5,7]	cpt[5,8]	cpt[5,9]
dimanche	cpt[6,0]	cpt[6,1]	cpt[6,2]	...	cpt[6,7]	cpt[6,8]	cpt[6,9]

```
// Calcule et affiche la quantité vendue de 10 produits
// pour chaque jour de la semaine (de 0 : lundi à 6 : dimanche).
algorithm statistiquesVentesSemaine
|   cpt: array of 7×10 integers
|   initialiser(cpt)
|   for jour from 0 to 6                                // Pour chaque jour de la semaine
|   |   traiterStock1Jour(cpt, jour)
|   |   for produit from 0 to 9
|   |   |   print "quantité vendue de produit ", produit, " ce jour ", jour, " : ", cpt[jour,produit]
|   |   end
|   end
end
```

```
// Initialise le tableau d'entiers à 0
algorithm initialiser(entiers  $\uparrow$  : array of 7×10 integers)
|   for i from 0 to 6
|   |   for j from 0 to 9
|   |   |   entiers[i,j] = 0
|   |   end
|   end
end
```

```
// Effectue le traitement du stock pour une journée.
algorithm traiterStock1Jour(cpt  $\uparrow$  : array of 7×10 integers, jour  $\downarrow$  : integer)
|   numéroProduit, quantité: integers
|   numéroProduit = ask "Introduisez le numéro du produit :"
|   while numéroProduit  $\geq$  0 et numéroProduit < 10
|   |   quantité = ask "Introduisez la quantité vendue :"
|   |   cpt[jour,numéroProduit] = cpt[jour,numéroProduit] + quantité
|   |   numéroProduit = ask "Introduisez le numéro du produit :"
|   end
end
```

1.2.5 Exercices

Exercice 1

Case nulle ?

Écrire un algorithme qui reçoit un tableau d'entiers (à n lignes et m colonnes) ainsi que les coordonnées d'une case (ligne, colonne) et qui retourne un booléen indiquant si la case désignée contient ou pas la valeur nulle.

```
algorithm estNul(tab: array of  $n \times m$  integers, lg, col: integers)  $\rightarrow$  boolean
```

Exercice 2

Assigner une case

Écrire un algorithme qui reçoit un tableau d'entiers (à n lignes et m colonnes) ainsi que les coordonnées d'une case (ligne, colonne) et une valeur entière. L'algorithme met la valeur donnée

dans la case indiquée pour autant que la case contienne actuellement la valeur nulle. Dans le cas contraire, l'algorithme ne fait rien.

```
algorithm assigner(tab ↑ : array of n × m integers, lg ↓, col ↓, val ↓ : integers)
```

Exercice 3

Un bord du tableau

Écrire un algorithme qui reçoit un tableau d'entiers (à n lignes et m colonnes) ainsi que les coordonnées d'une case (ligne, colonne). L'algorithme doit indiquer si la case donnée est ou non sur un **bord** du tableau.

```
algorithm estBord(tab: array of n × m integers, lg, col: integers) → boolean
```

Exercice 4

Un coin du tableau

Écrire un algorithme qui reçoit un tableau d'entiers (à n lignes et m colonnes) ainsi que les coordonnées d'une case (ligne, colonne). L'algorithme doit indiquer si la case donnée est ou non sur un des 4 **coins** du tableau.

```
algorithm estCoin(tab: array of n × m integers, lg, col: integers) → boolean
```

1.3 La troisième dimension (et au-delà)

Certaines situations complexes nécessitent l'usage de tableaux à 3 voire plus de dimensions.



Pour déclarer un tableau statique à k dimensions, on écrira :

```
nomTableau: array of tailleDim1 × ... × tailleDimK TypeElément
```

1.4 Parcours d'un tableau à deux dimensions

Comme nous l'avons fait pour les tableaux à une dimension, envisageons le parcours des tableaux à deux dimensions (n lignes et m colonnes). Nos algorithmes sont valables quel que soit le type des éléments. Utilisons T pour désigner un type quelconque.

```
tab: array of n × m T
```

Commençons par des cas plus simples où on ne parcourt qu'une seule des dimensions puis attaquons le cas général.

1.4.1 Parcours d'une dimension

On peut vouloir ne parcourir qu'une seule ligne du tableau. Si on parcourt la ligne l , on visite les cases $(l, 0)$, $(l, 1)$, ..., $(l, m - 1)$. L'indice de ligne est constant et c'est l'indice de colonne qui varie.

l	1	2	3	4	5

Ce qui donne l'algorithme :

```
// Parcours de la ligne ligne d'un tableau à deux dimensions
algorithm affichageElémentsLigne(tab: array of  $n \times m$  T, ligne: integer)
|   for colonne from 0 to m-1
|   |   print tab[ligne,colonne]           // On peut faire autre chose qu'afficher
|   end
end
```

Retenons : pour parcourir une ligne, on utilise une boucle sur les colonnes.

Symétriquement, on pourrait considérer le parcours d'une colonne avec l'algorithme suivant.

```
// Parcours de la colonne colonne d'un tableau à deux dimensions
algorithm affichageElémentsColonne(tab: array of  $n \times m$  T, colonne: integer)
|   for ligne from 0 to n-1
|   |   print tab[ligne,colonne]           // On peut faire autre chose qu'afficher
|   end
end
```

Si le tableau est carré ($n = m$) on peut aussi envisager le parcours des deux diagonales.

Pour la diagonale descendante, les éléments à visiter sont $(0, 0)$, $(1, 1)$, $(2, 2)$, \dots , $(n-1, n-1)$.

1		
	2	
		3

Une seule boucle suffit comme le montre l'algorithme suivant.

```
// Parcours de la diagonale descendante d'un tableau carré
algorithm affichageElémentsDiagonaleDescendante(tab: array of  $n \times n$  T)
|   for i from 0 to n-1
|   |   print tab[i,i]           // On peut faire autre chose qu'afficher
|   end
end
```

Pour la diagonale montante, on peut envisager deux solutions, avec deux indices ou un seul en se basant sur le fait que $i + j = n - 1 \Rightarrow j = n - 1 - i$.

		1
	2	
3		

```
// Parcours de la diagonale montante d'un tableau carré - version 2 indices
algorithm affichageElémentsDiagonaleMontante(tab: array of  $n \times n$  T)
|   j: integer
|   j = n-1
|   for i from 0 to n-1
|   |   print tab[i,j]           // On peut faire autre chose qu'afficher
|   |   j = j - 1
|   end
end
```

```
// Parcours de la diagonale montante d'un tableau carré - version 1 indice
algorithm affichageElémentsDiagonaleMontante(tab: array of  $n \times n$  T)
|   for i from 0 to n-1
|   |   print tab[i, n - 1 - i]           // On peut faire autre chose qu'afficher
|   end
end
```

1.4.2 Parcours des deux dimensions

Parcours par lignes et par colonnes

Les deux parcours les plus courants sont les parcours ligne par ligne et colonne par colonne. Les tableaux suivants montrent dans quel ordre chaque case est visitée dans ces deux parcours.

Parcours ligne par ligne

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

Parcours colonne par colonne

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

Le plus simple est d'utiliser deux boucles imbriquées

```
// Parcours d'un tableau à 2 dimensions, ligne par ligne
algorithm affichageElémentsLigneParLigne(tab: array of  $n \times m$  T)
  for lg from 0 to n-1
    for col from 0 to m-1
      print tab[lg,col]           // On peut faire autre chose qu'afficher
    end
  end
end
```

```
// Parcours d'un tableau à 2 dimensions, colonne par colonne
algorithm affichageElémentsColonneParColonne(tab: array of  $n \times m$  T)
  for col from 0 to m-1
    for lg from 0 to n-1
      print tab[lg,col]           // On peut faire autre chose qu'afficher
    end
  end
end
```

Mais on peut obtenir le même résultat avec une seule boucle si l'indice sert juste à compter le nombre de passages ($n*m$) et que les indices de lignes et de colonnes sont gérés manuellement.

L'algorithme suivant montre ce que ça donne pour un parcours ligne par ligne. La solution pour un parcours colonne par colonne est similaire et laissée en exercice.

```
// Parcours d'un tableau à 2 dimensions via une seule boucle
algorithm affichageElémentsLigneParLigne(tab: array of  $n \times m$  T)
  lg, col: integers
  lg = 0
  col = 0
  for i from 1 to n*m
    print tab[lg,col]           // On peut faire autre chose qu'afficher
    col = col + 1                // Passer à la case suivante
    if col = m                  // On déborde sur la droite, passer à la ligne suivante
      col = 0
      lg = lg + 1
    end
  end
end
```

L'avantage de cette solution apparaîtra quand on verra des situations plus difficiles.

Interrompre le parcours

Comme avec les tableaux à une dimension, envisageons l'arrêt prématuré lors de la rencontre d'une certaine condition. Et, comme avec les tableaux à une dimension, transformons d'abord nos *pour* en *tant que*.

Par exemple, montrons les deux parcours ligne par ligne, avec une et deux boucle(s).

```
// Parcours d'un tableau à 2 dimensions, ligne par ligne, via un tant que
algorithm affichageElémentsLigneParLigne(tab: array of  $n \times m$   $T$ )
  lg, col: integers
  lg = 0
  while lg < n
    col = 0
    while col < m
      print tab[lg, col]           // On peut faire autre chose qu'afficher
      col = col + 1
    end
    lg = lg + 1
  end
end
```

```
// Parcours d'un tableau à 2 dimensions via une seule boucle et un tant que
algorithm affichageElémentsLigneParLigne(tab: array of  $n \times m$   $T$ )
  lg, col, i: integers
  lg = 0
  col = 0
  i = 1
  while i ≤  $n \times m$                                      // ou "lg < n"
    print tab[lg, col]           // On peut faire autre chose qu'afficher
    col = col + 1
    if col = m                                           // Passer à la case suivante
      col = 0
      lg = lg + 1
    end
    i = i + 1
  end
end
```

On peut à présent introduire le test comme on l'a fait dans les algorithmes de parcours des tableaux à une dimension.

Illustrons-le au travers de deux exemples où on cherche un élément particulier. Le premier introduit un test en utilisant un booléen alors que le second introduit un test sans utiliser de booléen. Dans les deux cas, l'algorithme retourne un booléen indiquant si l'élément a été trouvé ou pas.

1.4. PARCOURS D'UN TABLEAU À DEUX DIMENSIONS

```
// Parcours avec test d'arrêt - deux boucles et un booléen
algorithm chercherElément(tab: array of  $n \times m$  T, élt: T)  $\rightarrow$  boolean
  lg, col: integers
  trouvé: boolean
  trouvé = faux
  lg = 0
  while lg < n ET NON trouvé
    col = 0
    while col < m ET NON trouvé
      if tab[lg, col] = élt
        trouvé = vrai
      else
        col = col + 1
      end
    end
    if NON trouvé
      lg = lg + 1
    end
  end
  return trouvé
end
```

```
// Parcours avec test d'arrêt - une boucle et pas de booléen
algorithm chercherElément(tab: array of  $n \times m$  T, élt: T)
  lg, col, i: integers
  lg = 0
  col = 0
  i = 1
  while  $i \leq n * m$  ET tab[lg, col]  $\neq$  élt
    col = col + 1
    if col = m
      col = 0
      lg = lg + 1
    end
    i = i + 1
  end
  return  $i < n * m$ 
end
```

Parcours plus compliqué - le serpent

Envisageons un parcours plus difficile illustré par le tableau suivant.

1	2	3	4	5
10	9	8	7	6
11	12	13	14	15

Le plus simple est d'adapter l'algorithme de parcours avec une seule boucle en introduisant un sens de déplacement, ce qui donne l'algorithme :

```
// Parcours du serpent dans un tableau à deux dimensions
algorithm affichageElémentsSerpent(tab: array of  $n \times m$  T)
| lg, col, depl: integers
| lg = 0
| col = 0
| depl = 1                                     // 1 pour avancer, -1 pour reculer
| for i from 1 to  $n*m$ 
| | print tab[lg, col]                       // On peut faire autre chose qu'afficher
| | if  $0 \leq col + depl$  ET  $col + depl < m$ 
| | | col = col + depl                         // On se déplace dans la ligne
| | else
| | | lg = lg + 1                             // On passe à la ligne suivante
| | | depl = -depl                           // et on change de sens
| | end
| end
end
```

1.5 Exercices

Exercice 5 Affichage

Écrire un algorithme qui affiche tous les éléments d'un tableau (à n lignes et m colonnes) ligne par ligne.



Écrivez un autre algorithme qui affiche cette fois les éléments colonne par colonne

Exercice 6 Cases adjacentes

Écrire un algorithme qui reçoit un tableau d'entiers (à n lignes et m colonnes) ainsi que les coordonnées d'une case (ligne, colonne) et *affiche* les coordonnées des cases *adjacentes*.

Exercice 7 Les nuls

Écrire un algorithme qui reçoit un tableau ($n \times m$) d'entiers et qui retourne la proportion d'éléments nuls dans ce tableau.



Exercice 8 Le tableau de cotes

Soit un tableau à n lignes et m colonnes d'entiers où une ligne représente les notes sur 20 d'un étudiant et les colonnes toutes les notes d'un cours.

Écrire un algorithme recevant ce tableau en paramètre et retournant le pourcentage d'étudiants ayant obtenu une moyenne supérieure à 50%.

Exercice 9 Le triangle de Pascal

Le triangle de Pascal est construit de la façon suivante :



- ▷ la ligne initiale contient un seul élément de valeur 1 ;
- ▷ chaque ligne possède un élément de plus que la précédente ;
- ▷ chaque ligne commence et se termine par 1 ;
- ▷ pour calculer un nombre d'une autre case du tableau, on additionne le nombre situé dans la case située juste au-dessus avec celui dans la case à la gauche de la précédente.

Écrire un algorithme qui reçoit en paramètre un entier n , et qui renvoie un tableau contenant les $n + 1$ premières lignes du triangle de Pascal (indiquées de 0 à n).

N.B. : le « triangle » sera bien entendu renvoyé dans un tableau carré (ce qui ne sera forcément le cas en Java). Quid des cases non occupées ?

Par exemple, pour n qui vaut 5, on aura le tableau ci-contre.

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

Exercice 10 Tous positifs



Écrire un algorithme qui reçoit un tableau ($n \times m$) d'entiers et qui vérifie si tous les nombres qu'il contient sont strictement positifs. Bien sûr, on veillera à éviter tout travail inutile ; la rencontre d'un nombre négatif ou nul doit arrêter l'algorithme.

Exercice 11 Toute une ligne de valeurs non nulles ?

Écrire un algorithme qui reçoit un tableau d'entiers (à n lignes et m colonnes) ainsi qu'un numéro de ligne et qui retourne un booléen indiquant si la ligne donnée du tableau ne contient que des valeurs non nulles.

algorithm lignePleine(tab: array of $n \times m$ integers, lg: integer) \rightarrow boolean

Faites de même pour une colonne.

Exercice 12 Le carré magique



Un carré magique est un tableau d'entiers carré (c'est-à-dire possédant autant de lignes que de colonnes) ayant la propriété suivante : si on additionne les éléments d'une quelconque de ses lignes, de ses colonnes ou de ses deux diagonales, on obtient à chaque fois le même résultat.

Écrire un algorithme recevant en paramètres le tableau ($n \times n$) d'entiers représentant le carré et renvoyant une valeur booléenne indiquant si c'est un carré magique ou pas.

Exercice 13 Lignes et colonnes

Écrire un algorithme qui reçoit un tableau d'entiers à 2 dimensions en paramètre et qui retourne un booléen indiquant si ce tableau possède 2 lignes ou 2 colonnes identiques.

Dans l'affirmative, cet algorithme renverra également en paramètres les informations suivantes :

- ▷ les indices des lignes ou colonnes identiques
- ▷ un caractère valant 'L' ou 'C' selon qu'il s'agit de lignes ou de colonnes

Dans la négative, les valeurs de ces paramètres seront indéterminées ou quelconques, elles ne seront de toute façon pas utilisées par l'algorithme appelant.

Exercice 14 Le contour du tableau

On donne un tableau d'entiers **tabEnt** à n lignes et m colonnes. Écrire un algorithme retournant la somme de tous les éléments *impairs* situés sur le bord du tableau.

Exemple : pour le tableau suivant, l'algorithme doit renvoyer 32

3	4	6	11
2	21	7	9
1	5	12	3

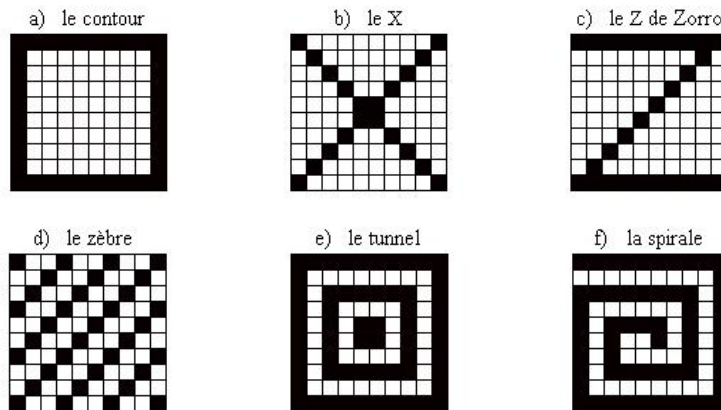
Et pour le suivant, l'algorithme doit renvoyer 6

4	1	2	8	5
---	---	---	---	---

Exercice 15 À vos pinceaux !

On possède un tableau à n lignes et n colonnes dont les éléments de type Couleur valent NOIR ou BLANC. On suppose que le tableau est initialisé à BLANC au départ. Écrire un algorithme

qui *noircit* les cases de ce tableau comme le suggèrent les dessins suivants (les exemples sont donnés pour un tableau 10 x 10 mais les algorithmes doivent fonctionner quelle que soit la taille du tableau).



Notes

- ▷ Le zèbre doit toujours présenter des lignes obliques et parallèles, quelle que soit la taille.
- ▷ La spirale est un véritable défi et vous est donné comme exercice facultatif. Ne le faites pas si vous êtes en retard.

Exercice 16

Exercices sur la complexité

Quelle est la complexité

- a) d'un algorithme de parcours d'un tableau $n \times n$?
- b) des algorithmes que vous avez écrits pour les exercices : "Les nuls", "Tous positifs", "Le carré magique" et "Le contour d'un tableau" ?
- c) des algorithmes que vous avez écrits pour résoudre les exercices du pinceau ?

Le cours de Java vous a présenté la programmation orienté objet. Dans ce chapitre, nous allons rapidement revoir ce sujet et présenter comment nous allons l'utiliser dans ce cours. Nous nous contenterons de parler d'*encapsulation*. Les autres piliers de l'orienté objet (*héritage* et *polymorphisme*) ne seront pas vus cette année.



2.1 Motivation

Au cours de Java, vous avez vu que l'orienté objet permet de structurer une application en regroupant dans un même *objet* des données et le code qui va manipuler ces données.

Une autre façon de voir l'orienté objet est de constater qu'une classe permet de définir un nouveau *type de données*. La notion de *structure* permet déjà cela mais de façon limitée car elle ne reprend que des données et pas du code. Avec l'orienté objet, on dispose de méthodes définissant ce qu'on peut faire avec des données (les objets) de ce type. C'est ainsi que nous l'utiliserons pour définir les listes dans un prochain chapitre.

2.2 Illustration : une durée

Voyons tout cela au travers d'un exemple complet. Il est parfois utile d'avoir à sa disposition un type de données permettant de représenter une durée. Utiliser plusieurs entiers (un pour les heures, un autre pour les minutes, un autre encore pour les secondes) n'est pas pratique. Utiliser une structure est déjà mieux mais offre moins d'avantage que l'orienté objet. Voyons comment définir ce nouveau type de données en orienté objet.

2.2.1 Ce que l'on veut vraiment

Avant tout, il faut bien préciser ce que l'on veut décrire et bien faire la distinction entre un *moment* et une *durée*. L'« heure » est un concept multifacette. Parle-t-on de l'heure comme moment dans la journée ou de l'heure comme représentant une durée ? Dans le premier cas, elle ne peut dépasser 24h et la différence entre 2 heures n'a pas de sens (ou plus précisément n'est pas une heure, mais une durée !). Ce que nous nous proposons de créer ici est une durée, correspondant au deuxième cas. Et pour être plus précis encore, nous allons nous limiter à une précision à la seconde près, pas plus¹.

1. Ajouter plus de précision ne serait pas plus compliqué à faire.

2.2.2 Le comportement (les méthodes)

La première question à se poser est celle des services que l'on veut fournir, c'est-à-dire des méthodes publiques de la classe. On doit pouvoir *construire* une durée. On doit pouvoir connaître le nombre de jours, d'heures, minutes ou secondes correspondant à une durée. On doit pouvoir effectuer des calculs avec des durées (addition, soustraction). Enfin, on doit pouvoir comparer des durées. Arrêtons-nous là, mais en pratique, on pourrait trouver encore bon nombre d'autres méthodes qu'il serait intéressant de fournir.

Voici comment nous allons noter tout cela au cours d'algorithmique.

```
class Durée
  public :
    constructor Durée(secondes : entier)
    constructor Durée(heure, minute, seconde : entiers)

    method getJour() → entier // nb de jours dans une durée

    method getHeure() → entier // entier entre 0 et 23 inclus
    method getMinute() → entier // entier entre 0 et 59 inclus
    method getSeconde() → entier // entier entre 0 et 59 inclus

    method getTotalHeures() → entier // Le nombre total d'heures
    method getTotalMinutes() → entier // Le nombre total de minutes
    method getTotalSecondes() → entier // Le nombre total de secondes

    method ajouter(autreDurée : Durée)
    method différence(autreDurée : Durée) → Durée
    method plusPetit(autreDurée : Durée) → booléen
end
```



Quelques remarques

- ▷ On a deux constructeurs, ce qui offre plus de souplesse pour initialiser un objet. On parle de « **surcharge** » des constructeurs.
- ▷ Faisons bien la distinction entre les méthodes `getXXX()` et `getTotalXXX()`. Par exemple, la méthode `getMinute()` retourne la valeur de la composante « minutes » dans une représentation HMS tandis que la méthode `getTotalMinutes()` retourne le nombre total de minutes entières pour cette durée. Ex : pour 1h23'12", `getMinute()` retourne 23 et `getTotalMinutes()` retourne 83. Idem avec les heures et les secondes.
- ▷ Les méthodes `getTotalXXX()` retournent le nombre (toujours entier) de XXX contenus dans la durée. Exemple, avec la durée 0h23'52", `getTotalMinutes()` retourne 23 et pas 24 (autrement dit, il n'y a pas d'arrondi vers le haut).
- ▷ Il n'y a pas de *mutateur* (`setXXX()`). Ce qui signifie qu'on ne peut pas changer directement la valeur de l'objet après son initialisation. Les seules modifications viendront de la méthode `ajouter()`. On aurait pu définir des mutateurs mais nous n'avons pas jugé utile de le faire dans ce cas précis. Vous verrez dans le cours de Java des motivations à ce choix.
- ▷ La méthode `ajouter()` ne retourne rien. En effet, elle ajoute la durée à l'objet sur lequel est appelée la méthode. C'est un choix ; on aurait aussi pu dire que la méthode ne modifie pas l'objet mais en retourne un autre qui représente la somme. Dans ce cas, on l'aurait plutôt appelée « `plus()` ».
- ▷ La méthode `différence()`, elle, renvoie toujours une durée (positive).
- ▷ Nous ne définissons pas de méthode d'affichage similaire au `toString()` qu'on retrouve en Java. L'affichage correct de l'information ne fait pas partie des préoccupations de ce cours. On supposera que "**afficher objet**" affiche correctement les données associées à l'objet.

2.2.3 La représentation de l'état (les attributs)

La question suivante est : « Comment représenter une durée en interne ? ». Plusieurs possibilités existent. Par exemple :

- ▷ via le nombre d'heures, de minutes et de secondes
- ▷ via le nombre total de secondes
- ▷ via une chaîne, par exemple au format « HH : MM : SS » où HH pourrait éventuellement excéder 23.

Le premier choix semble le plus évident mais réfléchissons-y de plus près. D'une part, pourquoi se limiter aux heures. On pourrait introduire un champ 'jour' (après tout on a bien une méthode `getJour()`). Quel critère doit vraiment nous permettre de décider ? Il faut une représentation qui soit suffisante (tout est représenté) et qui permette d'écrire des méthodes lisibles et si possible efficaces (c'est-à-dire où le calcul est rapide). Selon ces critères, la deuxième représentation est de loin la meilleure.

Voilà comment nous indiquons les attributs d'une classe.

```
class Durée
|   private :
|       totalSecondes: entier
|   public :
|       // Ici viennent les constructeurs et les méthodes
end
```

Pour rappel de votre cours de langage, ce qui est privé n'est pas utilisable directement par du code extérieur à la classe. Un code extérieur, manipulant des objets de cette classe, ne peut utiliser que ce qui est public.

2.2.4 L'implémentation

On est à présent prêt pour écrire le code des méthodes. Pour une meilleure lisibilité, nous gardons les signatures des méthodes dans la classe et nous détaillons leur contenu en dehors. Ce qui donne :

```
class Durée
|   private :
|       totalSecondes: entier
|   public :
|       constructor Durée(secondes : entier)
|       constructor Durée(heure, minute, seconde : entiers)
|
|       method getJour() → entier                // nb de jours dans une durée
|       method getHeure() → entier                // entier entre 0 et 23 inclus
|       method getMinute() → entier               // entier entre 0 et 59 inclus
|       method getSeconde() → entier              // entier entre 0 et 59 inclus
|
|       method getTotalHeures() → entier           // Le nombre total d'heures
|       method getTotalMinutes() → entier         // Le nombre total de minutes
|       method getTotalSecondes() → entier        // Le nombre total de secondes
|
|       method ajouter(autreDurée : Durée)
|       method différence(autreDurée : Durée) → Durée
|       method plusPetit(autreDurée : Durée) → booléen
end
```

2.2. ILLUSTRATION : UNE DURÉE

```
constructor Durée(secondes : entier)
| if secondes < 0
| | error "paramètre négatif"a
| end
| totalSecondes = secondes
end

constructor Durée(heure, minute, seconde : entiers)
| if heure < 0 OU minute < 0 OU seconde < 0 OU minute>59 ou seconde>59
| | error "un des paramètres est invalide"
| end
| totalSecondes = 3600*heure + 60*minute + seconde
end

// Retourne le nombre de jours dans une représentation JJ/HH :MM :SS
method getJour() → entier
| return totalSecondes DIV (3600*24)
end

// Retourne le nombre d'heures dans une représentation JJ/HH :MM :SS
method getHeure() → entier
| return (totalSecondes DIV 3600) MOD 24 // On doit enlever les jours éventuels
end

// Retourne le nombre de minutes dans une représentation JJ/HH :MM :SS
method getMinute() → entier
| return (totalSecondes DIV 60) MOD 60 // On doit enlever les heures éventuelles
end

// Retourne le nombre de secondes dans une représentation JJ/HH :MM :SS
method getSeconde() → entier
| return totalSecondes MOD 60 // On doit enlever les minutes éventuelles
end

// Retourne le nombre entier d'heures complètes
method getTotalHeures() → entier
| return totalSecondes DIV 3600
end

// Retourne le nombre entier de minutes complètes
method getTotalMinutes() → entier
| return totalSecondes DIV 60
end

// Retourne le nombre entier de secondes complètes
method getTotalSecondes() → entier
| return totalSecondes
end

method ajouter(autreDurée : Durée)
| totalSecondes = totalSecondes + autreDurée.getTotalSecondes()
end

method différence(autreDurée : Durée) → Durée
| return new Durée(valeurAbsolue(totalSecondes - autreDurée.getTotalSecondes()))
end

method plusPetit(autreDurée : Durée) → booléen
| return totalSecondes < autreDurée.getTotalSecondes()
end
```

^a. L'instruction **error** indique que l'algorithme ne peut pas poursuivre normalement. Il s'arrête avec un message d'erreur.

2.2.5 Utilisation

Pour utiliser le nouveau type de donnée créé, il faut l'instancier, c'est-à-dire créer un nouvel objet de ce type. Nous allons utiliser le mot clé **new** pour rester très proche de Java.

Illustrons cela au travers d'un petit algorithme qui calcule la différence entre deux durées.

```

algorithm diffDurée
  durée1, durée2: Durée                                // Les variables sont déclarées
  durée1 = new Durée(3, 4, 49)                        // durée1 est créé
  durée2 = new Durée(3, 24, 37)                        // durée2 est créé
  print durée2.différence(durée1)
end

```

2.3 Quelques éléments de syntaxe

Clarifions certaines notations liées aux objets.

- ▷ Pour un attribut `brol`, on choisira de nommer l'accesseur² `getBrol` et le mutateur³ `setBrol`. Dans le cas particulier d'un attribut booléen, on pourra appeler l'accesseur `isBrol` ou encore `estBrol`.
- ▷ On peut directement afficher un objet. Cela affiche l'état d'un objet d'une façon claire pour l'utilisateur⁴.

```

rendezVous: Durée
rendezVous = new Durée(14, 23, 56)
print rendezVous                                // affichera 14, 23 et 56 dans un format lisible.

```

- ▷ De même, on peut directement lire un objet, ce qui a pour effet de créer un objet avec un état correspondant aux valeurs lues pour ses attributs.

```

rendezVous: Durée
read rendezVous

```

- ▷ La comparaison de deux objets est toujours un problème délicat en orienté objet. Nous nous baserons ici sur les conventions JAVA et utiliserons la notation `o1.égale(o2)` quand il s'agira de vérifier que `o1` et `o2` sont dans le même état, c'est-à-dire que leurs attributs ont la même valeur.
- ▷ Lorsqu'on déclare un objet, il n'est pas encore créé. On peut utiliser la valeur spéciale « rien » pour indiquer ou tester qu'un objet n'est pas encore créé.

```

parcours: Durée                                // parcours = rien
parcours = new Durée( 14, 23, 56 )              // parcours ≠ rien
if parcours ≠ rien
|   parcours = rien                            // parcours = rien
end

```

- ▷ Si une classe ne propose pas de constructeur, on peut néanmoins instancier un objet (via `new NomClasse()`). On considère dans ce cas que les attributs ne sont pas initialisés.

2. Pour rappel, un *accesseur* est une méthode donnant la valeur d'un attribut.

3. Pour rappel, un *mutateur* est une méthode permettant de modifier la valeur d'un attribut.

4. Le format précis n'est pas spécifié car il n'est pas important pour ce cours.

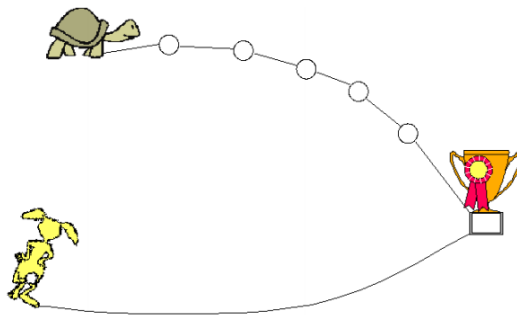
2.4 Mise en pratique : le lièvre et la tortue

Partons d'un petit jeu, « Le lièvre et la tortue »⁵, et voyons comment le coder en OO.

2.4.1 Description du jeu

Un lièvre et une tortue font une course. Le lièvre est plus rapide que la tortue. Pour donner plus de chance à la tortue de gagner une course de 6 km, on adopte la règle de jeu suivante :

- ▷ On lance un dé.
- ▷ Si le 6 sort, le lièvre est autorisé à démarrer et gagne la course ; sinon on laisse la tortue avancer d'un kilomètre.
- ▷ On recommence le procédé jusqu'à la victoire du lièvre ou de la tortue.



2.4.2 Solution non orientée objet

Pour ne pas aller trop vite et vous perdre tout de suite, commençons par une version non orientée objet du jeu.

Représenter le jeu

Il faut d'abord se poser cette question : Comment représenter le jeu ? Une représentation du jeu doit être complète. C'est-à-dire qu'à partir de cette représentation, on doit pouvoir indiquer exactement où on en est dans le jeu et pouvoir le poursuivre.

Pour le dire autrement, imaginons qu'on joue à ce jeu « en vrai », sur une table de jeu. La représentation informatique doit capturer tout ce qui est pertinent dans le jeu physique de sorte que, si on range la boîte de jeu, on peut, le lendemain, reconstruire le jeu exactement comme il était.

Dans notre exemple, cela veut dire quoi ?

La tortue

On doit pouvoir savoir où elle en est dans son avancée. Un petit entier (de 0 à 6) reprenant son avancée en km suffit. Appelons-le `avancéeTortue` par exemple.

Le lièvre

Pendant le jeu, il est en permanence au départ et lorsque sort un 6, il atteint directement l'arrivée. Plusieurs possibilités s'offrent à nous :

1. On pourrait imaginer un entier valant 0 ou 6, appelé `avancéeLièvre`.
2. On pourrait aussi imaginer un booléen à vrai lorsqu'il est au départ et faux lorsqu'il est à l'arrivée. On pourrait l'appeler `lièvreAuDépart`.

5. Lu sur le net : http://mathemathieu.free.fr/2b/doc/pb_algo/problemes_et_algorithmique.pdf (site désormais inaccessible)

3. Ou encore un booléen ayant exactement le sens inverse. Le nom devra être choisi judicieusement pour ne pas induire le lecteur en erreur. Par exemple, ici, on pourrait choisir `lièvreArrivé`.

Ici, le troisième choix nous semble le plus pertinent.

Le dé

Un entier de 1 à 6 suffit pour représenter le résultat d'un dé. Appelons-le simplement `dé`.

Les joueurs

Si on observe un jeu physique, on peut s'attarder sur les personnes en train de jouer. Faut-il les représenter ? On pourrait imaginer de connaître leur nom, le nombre de fois qu'elles ont joué à ce jeu, leur nombre de victoires. Dans l'énoncé, rien n'indique qu'il faille tenir compte de tout cela. On s'intéresse au jeu proprement dit, et c'est tout.

Le plateau

On peut imaginer que dans le jeu physique, il y aurait une sorte de plateau avec des km indiqués sur lequel avancerait la tortue. Mais il n'y a aucune information changeante sur ce plateau qui vaille la peine d'être retenue.

Un macro algorithme

Avant de se lancer dans l'écriture d'une solution détaillée du jeu, commençons par une solution non détaillée et voyons si tout semble clair et faisable.

```

algorithm jeuLièvreTortue
  Initialiser le jeu
  while le jeu n'est pas fini
    Lancer le dé
    if le dé est 6
      Le lièvre est arrivé
    else
      La tortue avance
    end
    print l'état du jeu
  end
  print le vainqueur
end

```

Détailler l'algorithme

Repassons à présent sur l'algorithme et vérifions que nous pouvons détailler chacun des points restés généraux.

- ▷ **Initialiser le jeu.** Il suffit de placer la tortue en 0 et d'indiquer que le lièvre n'est pas encore arrivé. La valeur initiale du dé n'a pas d'importance.
- ▷ **Le jeu n'est pas fini.** Le jeu sera fini lorsque le lièvre sera arrivé (ce qu'on peut tester grâce au booléen `lièvreArrivé`) ou que la tortue sera au km 6.
- ▷ **Lancer le dé.** C'est trivial si on utilise l'algorithme `hasard()` à notre disposition.
- ▷ **Le lièvre est arrivé.** Il suffit de mettre le booléen `lièvreArrivé` à vrai.
- ▷ **La tortue avance.** C'est trivial.
- ▷ **Afficher l'état du jeu.** À savoir, sur quelle face est tombé le dé et où se trouvent à présent le lièvre et la tortue.
- ▷ **Afficher le vainqueur.** Ce sera le lièvre si son booléen est à vrai et la tortue sinon (dans ce cas, son avancée sera forcément de 6 puisque le jeu est fini).

Au final, on obtient :

```

algorithm jeuLièvreTortue
  avancéeTortue: integer
  lièvreArrivé: boolean
  dé: integer

  avancéeTortue = 0
  lièvreArrivé = faux

  while avancéeTortue < 6 ET NON lièvreArrivé
    dé = hasard(6)
    if dé = 6
      | lièvreArrivé = vrai
    else
      | avancéeTortue = avancéeTortue + 1
    end
    print dé, avancéeTortue, lièvreArrivé
  end

  if lièvreArrivé
    | print "Le lièvre a gagné"
  else
    | print "Le tortue a gagné"
  end
end

```

2.4.3 Solution orientée objet

Voyons à présent ce que ça pourrait donner si on introduit de l'orienté objet. Examinons d'abord les objets physiques du jeu.

La tortue

On peut envisager de définir une classe pour la tortue. Une tortue a une avancée. Au départ, elle est au kilomètre 0. Elle peut avancer d'un kilomètre à la fois. Elle a fini et gagne lorsqu'elle arrive au kilomètre 6.

```

class Tortue
  private :
    | avancée: integer
  public :
    constructor Tortue
      | avancée = 0
    end

    method avancer
      | avancée = avancée + 1
    end

    method estArrivée() → boolean
      | return avancée = 6
    end

    method getAvancée() → integer
      | return avancée
    end
end

```

Remarquez qu'on n'introduit pas de mutateur car on veut que la tortue n'avance qu'en respectant les règles du jeu.

Le lièvre

On peut appliquer la même démarche pour le lièvre qui aurait un attribut booléen indiquant s'il est arrivé ou pas.

Le dé

Le dé est également un objet de notre jeu et peut être défini via une classe.

```
class Lièvre
|
| private :
|   arrivé: boolean
| public :
|   constructor Lièvre
|   |   arrivé = faux
|   end
|
|   method avancer
|   |   arrivé = vrai
|   end
|
|   method estArrivé() → boolean
|   |   return arrivé
|   end
|
end
```

```
class Dé
|
| private :
|   valeur: integer
| public :
|   constructor Dé
|   end
|
|   method lancer
|   |   valeur = hasard(6)
|   end
|
|   method getValeur() → integer
|   |   return valeur
|   end
|
end
```

L'algorithme du jeu

L'algorithme du jeu peut être réécrit en utilisant les trois classes qu'on vient de définir.

```
algorithm jeuLièvreTortue
|
| tortue: Tortue
| lièvre: Lièvre
| dé: Dé
|
| tortue = new Tortue()
| lièvre = new Lièvre()
| dé = new Dé()
|
| while NON tortue.estArrivée() ET NON lièvre.estArrivé()
| | dé.lancer()
| | if dé.getValeur()=6
| | | lièvre.avancer()
| | else
| | | tortue.avancer()
| | end
| | print dé.getValeur(), tortue.getAvancée(), lièvre.estArrivé()
| end
|
| if lièvre.estArrivé()
| | print "Le lièvre a gagné"
| else
| | print "Le tortue a gagné"
| end
|
end
```

Est-ce une bonne idée d'avoir défini ces trois classes ? C'est une question qu'il est légitime de se poser quand les classes sont aussi simples. Remarquons toutefois que le code est plus modulaire et que la méthode principale est plus facile à lire.

La classe Dé se justifie d'autant plus qu'elle pourra probablement servir à de nombreuses occasions. Ce sera encore plus le cas si on la généralise à des dés qui n'ont pas forcément 6 faces.

```

class Dé
  private :
    nbFaces: integer
    valeur: integer
  public :
    constructor Dé(nf: integer)
      | nbFaces = nf
    end

    method lancer
      | valeur = hasard(nbFaces)
    end

    method getValeur() → integer
      | return valeur
    end
end

```

Dans l'algorithme principal, le seul changement est la création du dé qui devient :

```

dé = new Dé(6)

```

2.4.4 Solution MVC (« Modèle-Vue-Contrôleur »)

Dans la version OO qu'on vient de voir, on a introduit trois classes mais il reste tout un morceau, l'algorithme principal, qui n'est pas OO. Peut-on aller plus loin dans l'OO ? Bien sûr ! Mais il y a de bonnes et de mauvaises façons de le faire.

La mauvaise approche est de simplement mettre l'algorithme principal dans une classe. Ce qui donnerait :

```

class LièvreTortue
  constructor LièvreTortue
  end

  method jouer
    | // Idem algorithme jeuLièvreTortue() ci-avant
  end
end

```

Ce qui réduirait l'algorithme principal à :

```

algorithm jeuLièvreTortue
  | jeu: LièvreTortue
  | jeu = new LièvreTortue()
  | jeu.jouer()
end

```

ou même, en se passant de la variable locale :

```

algorithm jeuLièvreTortue
  | (new LièvreTortue()).jouer()
end

```

Cette approche est correcte mais n'exploite en rien les avantages de l'OO. Une meilleure idée est de suivre l'approche MVC que nous allons vous expliquer.

Dans l'approche MVC, on découpe le code en différentes parties. La partie « modèle » regroupe les bouts de code qui font vraiment quelque chose (on parle de « métier ») tandis que la partie « vue » regroupe les bouts de code qui interagissent avec l'utilisateur (demandes et affichages). La partie « contrôleur », quant à elle, conserve le code qui fait le lien entre le modèle et la vue.

Si on respecte cette approche, le métier ne contient **aucune** interaction avec l'utilisateur et la vue ne s'occupe **que** de l'interaction avec l'utilisateur. Il y a là de nombreux avantages :

- ▷ Les compétences pour écrire le modèle (connaissance du métier, accès à des bases de données...) et le dialogue avec les utilisateurs (ergonomie, graphisme...) ne sont pas les mêmes. On peut donc confier ces parties à des équipes spécialisées.
- ▷ On pourra facilement changer le dialogue avec l'utilisateur. Ainsi, si on possède une version console du jeu, il suffira, pour en faire une autre version (console, graphique, web...) de recommencer la vue (et probablement d'adapter le contrôleur) sans toucher au modèle.

Le modèle. Dans notre exemple, le modèle contiendrait les algorithmes suivants :

- ▷ Initialiser le jeu : placer la tortue et le lièvre à leur position de départ.
- ▷ Jouer un coup : lancer le dé et déplacer le lièvre ou la tortue.
- ▷ Tester si le jeu est fini ou pas.
- ▷ Trouver le vainqueur.

Chacun de ces algorithmes est implémenté par une méthode (sauf l'initialisation qui est du ressort du constructeur). Au niveau des attributs, on retrouve les éléments du jeu : le lièvre, la tortue et le dé.⁶

Ce qui donne :

```
class LièvreTortue
|   private :
|       tortue: Tortue
|       lièvre: Lièvre
|       dé: Dé
|   public :
|       constructor LièvreTortue
|       method estFini() → boolean
|       method jouerCoup
|       method getVainqueur() → string
|       //
|       + les accesseurs (getLièvre(), getTortue() et getDé()) des attributs mais pas les mutateurs
end
```

```
constructor LièvreTortue
|   tortue = new Tortue()
|   lièvre = new Lièvre()
|   dé = new Dé(6)
end

method estFini() → boolean
|   return tortue.estArrivée() OU lièvre.estArrivé()
end
```

6. Une autre façon de voir les choses est de dire qu'on trouve en attributs les variables locales de la version non OO qui sont partagées par les différentes méthodes. Ici, il s'agit de toutes les variables locales mais ce n'est pas toujours le cas. Le dé, par exemple, n'est un attribut que parce que la vue voudra le connaître pour le montrer à l'utilisateur. Sans cela, il pourrait être une variable locale de la méthode jouerCoup().

```

method jouerCoup
  dé.lancer()
  if dé.getValeur()==6
  |   lièvre.avancer()
  end
  tortue.avancer()
end

method getVainqueur() → string
  if lièvre.estArrivé()
  |   return "Lièvre"
  else if tortue.estArrivée()
  |   return "Tortue"
  else
  |   return "Aucun, partie en cours"
  end
end

```

Remarque : Dans un langage comme JAVA, la méthode `getVainqueur()` pourrait lancer une `IllegalStateException` si elle est appelée avant que la partie ne soit finie.

La vue. Avec notre exemple, plutôt simple, il n'y a pas de lecture mais il reste des affichages :

- ▷ Afficher l'état du jeu après un coup : valeur du dé et nouvelles positions du lièvre et de la tortue.
- ▷ Afficher le vainqueur.

Cet exemple est probablement trop simple pour nécessiter une classe ⁷. On pourrait se contenter de deux algorithmes classiques.

```

algorithm afficherÉtat(jeu: LièvreTortue)
  print jeu.getDé().getValeur()
  print jeu.getTortue().getAvancée()
  print jeu.getLièvre().estArrivé()
end

algorithm afficherVainqueur(jeu: LièvreTortue)
  print "Le gagnant est : ", jeu.getVainqueur()
end

```

Le contrôleur. Le contrôleur est ce qui n'a pas été placé dans la vue ou le modèle, c'est-à-dire le code qui crée la dynamique entre tous ces éléments.

```

algorithm jeuLièvreTortue
  jeu: LièvreTortue
  jeu = new LièvreTortue()
  while NON jeu.estFini()
  |   jeu.jouerCoup()
  |   afficherÉtat(jeu)
  end
  afficherVainqueur(jeu)
end

```

Remarquez la concision et la lisibilité de ce qu'on vient d'écrire.

7. Une erreur classique est de placer ces algorithmes dans les classes associées du modèle car cela contrevient à la règle : aucune interaction utilisateur dans le modèle.

Imaginons que l'on désire manipuler par programme une liste de contacts ou encore une liste de rendez-vous. Cette liste va varier ; sa taille n'est donc pas fixée. Utiliser un tableau à cet effet n'est pas l'idéal. En effet, la taille d'un tableau ne peut plus changer une fois le tableau créé. Il faudrait le surdimensionner, ce qui n'est pas économe.



Il serait intéressant de disposer d'une structure qui offre toutes les facilités d'un tableau tout en pouvant « grandir » si nécessaire. Construisons une telle structure de données et appelons-la « List » pour rester en phase avec son appellation commune en Java.

Par exemple, considérons une liste de courses. On pourrait la représenter ainsi :

1. "fromage"
2. "pain"
3. "salami"

On pourrait ajouter un élément en fin de liste, par exemple de l'eau, pour obtenir la liste :

1. "fromage"
2. "pain"
3. "salami"
4. "eau"

On pourrait aussi supprimer un élément de la liste, par exemple le pain, et obtenir :

1. "fromage"
2. "salami"
3. "eau"

On pourrait aussi insérer un élément dans la liste, par exemple une baguette, ce qui décale, de facto, la position des suivants.

1. "fromage"
2. "salami"
3. "baguette"
4. "eau"

Et encore plein de choses que nous allons détailler.

3.1 La classe List

Intéressons-nous au comportement qu'on attend d'une liste, c'est-à-dire les méthodes qu'elle doit fournir. Ce comportement sera identique quel que soit le type des éléments de la liste ; une liste de chaînes et une liste d'entiers ne se distinguent que par le type de certains paramètres et valeurs de retour. Ici, nous indiquons *T* pour indiquer un type quelconque ; vous pouvez le remplacer par ce qui vous convient : entier, chaîne, Date...

```
// T est un type quelconque
class List of T
  public :
    constructor List of T                                // construit une liste vide
    method get(pos : integer) → T                        // donne un élément en position pos
    method set(pos : integer, valeur : T)                // modifie un élément en position pos
    method taille() → integer                             // donne le nombre actuel d'éléments
    method ajouter(valeur : T)                           // ajoute un élément en fin de liste
    method insérer(pos : integer, valeur : T)             // insère un élément en position pos
    method supprimer                                     // supprime le dernier élément
    method supprimerPos(pos : integer)                   // supprime l'élément en position pos
    method supprimer(valeur : T) → boolean               // supprime l'élément de valeur donnée
    method vider                                         // vide la liste
    method estVide() → boolean                           // la liste est-elle vide ?
    method existe(valeur ↓ : T, pos ↑ : entier) → boolean // recherche un élément
end
```

Quelques précisions s'imposent :

- ▷ Les méthodes « **get** » et « **set** » permettent de connaître ou modifier un élément de la liste. On considère, au cours d'algorithmique, que le premier élément de la liste est en position 0.
- ▷ « **ajouter** » ajoute un élément en fin de liste (elle grandit donc d'une unité)
- ▷ « **insérer** » insère un élément à une position donnée (entre 0 et *taille*-1). L'élément qui s'y trouvait est décalé d'une position ainsi que tous les éléments suivants.
- ▷ La méthode « **supprimerPos** » supprime un élément d'une position donnée en décalant les éléments suivants. On pourrait imaginer une technique plus rapide consistant à placer le dernier élément à la place de l'élément supprimé mais ce faisant on changerait l'ordre relatif des éléments ce qui va à l'encontre de l'idée intuitive qu'on se fait d'une liste. Cette amélioration pourrait plutôt s'envisager dans une structure de type **ensemble** pour lequel il n'y a pas d'ordre relatif entre les éléments.
- ▷ La version de « **supprimer** » avec une valeur en paramètre enlève un élément de valeur donnée. Elle retourne un booléen indiquant si la suppression a pu se faire ou pas (ce qui sera le cas si la valeur n'est pas présente dans la liste). Si la valeur existe en plusieurs exemplaires, on prendra la convention arbitraire que la méthode n'en supprime que la première occurrence.
- ▷ La méthode « **existe** » permet de savoir si un élément donné existe dans la liste.
 - ▷ si c'est le cas, elle précise aussi sa position dans le paramètre sortant **pos**
 - ▷ si l'élément n'existe pas, ce paramètre est indéterminé
 - ▷ si l'élément est présent en plusieurs exemplaires, la méthode donne la position de la première occurrence.
- ▷ En pratique, il serait intéressant de chercher un élément à partir d'une partie de l'information qu'elle contient mais c'est difficile à exprimer de façon générique c'est-à-dire lorsque le type n'est pas connu a priori.

Exemple : manipulations de base

Soit l'algorithme suivant :

```

algorithm ex1
  liste: List of integers
  liste = new List of integers()
  liste.ajouter(42)
  liste.ajouter(54)
  liste.set(1,44)
  liste.insérer(1,43)
  liste.supprimerPos(2)
  liste.supprimer(42)
  liste.vider
end

```

Après sa création, la liste est vide. Ensuite, elle passe par les états suivants :

0. 42	0. 42	0. 42	0. 42	0. 42	0. 43
	1. 54	1. 44	1. 43	1. 43	
			2. 44		

Enfin, le dernier appel la vide complètement

Exemple : afficher une liste

Écrivons l'algorithme qui affiche tous les éléments d'une liste, reçue en paramètre.

```

algorithm afficher(liste: List of integers)
  for i from 0 to liste.taille()-1
  |   print liste.get(i)
  end
end

```

Exemple : recherche du minimum

Dans le chapitre sur les tableaux, vous avez fait un exercice consistant à afficher tous les indices où se trouve le minimum d'un tableau. Reprenons-le et modifions-le afin qu'il retourne la liste des indices où se trouvent les différentes occurrences du minimum. On pourrait l'écrire ainsi :

```

algorithm indicesMinimum(tab: array of n integers) → List of integers
  min: integer
  indicesMin: List of integers
  min = tab[0]
  indicesMin = new List of integers()
  indicesMin.ajouter( 0 )
  for i from 1 to n-1
  |   if tab[i] = min
  |   |   indicesMin.ajouter( i )
  |   else if tab[i] < min
  |   |   indicesMin.vider()
  |   |   indicesMin.ajouter( i )
  |   |   min = tab[i]
  |   end
  |   // rien à faire si tab[i] > min
  end
  return indicesMin
end

```

3.2 Exercices

Exercice 1

Manipulation d'une liste

Écrire un algorithme qui crée la liste suivante :

- 0. 494
- 1. 209
- 2. 425

affiche sa taille, demande si la valeur 425 est présente, supprime la valeur 209 puis insère la valeur 101 en tête de liste.

Exercice 2

Liste des premiers entiers

Écrire un algorithme qui reçoit un entier n en paramètre et retourne la liste contenant les entiers de 1 à n dans l'ordre décroissant. On peut supposer que n est strictement positif.

Exercice 3

Somme d'une liste



Écrire un algorithme qui calcule la somme des éléments d'une liste d'entiers.

Exercice 4

Anniversaires

Écrire un algorithme qui reçoit une liste de structure `Personne` (nom + prénom + date de naissance) et retourne la liste de ceux qui sont nés durant un mois passé en paramètre (donné sous la forme d'un entier entre 1 et 12).

Exercice 5

Concaténation de deux listes



Écrire un algorithme qui reçoit 2 listes et ajoute à la suite de la première les éléments de la seconde ; la seconde liste n'est pas modifiée par cette opération.

Exercice 6

Le nettoyage

Écrire un algorithme qui reçoit une liste de chaînes en paramètre et supprime de cette liste tous les éléments de valeur donnée en paramètre. L'algorithme retournera le nombre de suppressions effectuées.

Exercice 7

Les extrêmes



Écrire un algorithme qui supprime le minimum et le maximum des éléments d'une liste d'entiers. On peut supposer que le maximum et le minimum sont uniques.

Exercice 8

Fusion de deux listes



Soit deux listes **triées** d'entiers (redondances possibles). Écrire un algorithme qui les fusionne. Le résultat est une liste encore triée contenant tous les entiers des deux listes de départ (qu'on laisse inchangées).

Exemple : Si les 2 listes sont (1, 3, 7, 7) et (3, 9), le résultat est (1, 3, 3, 7, 7, 9).

Exercice 9

Éliminer les doublons d'une liste



Soit une liste **triée** d'entiers avec de possibles redondances. Écrire un algorithme qui enlève les redondances de la liste.

Exemple : Si la liste est (1, 3, 3, 7, 8, 8, 8), le résultat est (1, 3, 7, 8).

- a) Faites l'exercice en créant une **nouvelle liste** (la liste de départ reste inchangée)
- b) Refaites l'exercice en **modifiant** la liste de départ (pas de nouvelle liste)

Exercice 10**Rendez-vous**

Soit la structure `RendezVous` composée d'une date (cf. la structure `Date` du cours de DEV1) et d'un motif de rencontre.

```
structure RendezVous  
|   date: Date  
|   motif: string  
end
```

Écrire un algorithme qui reçoit une liste de rendez-vous et la met à jour en supprimant tous ceux qui sont désormais passés.

Pour résoudre cet exercice, vous pouvez utiliser sans l'écrire un algorithme `aujourd'hui()` qui retourne la date du jour.

Les traitements de rupture

Dans ce chapitre, nous allons étudier une classe de problèmes qui peuvent tous se résoudre avec un même type d'algorithme : l'algorithme de rupture.

Considérons un problème comme celui-ci :

« Soit une liste d'étudiants, où un étudiant est représenté par un objet reprenant son nom, son numéro, son option et son année. Écrire un algorithme qui compte le nombre d'étudiants dans chaque section et, plus précisément, dans chaque année de chaque section. »

Nous verrons que l'algorithme de rupture sera adapté à la résolution de ce problème lorsque la liste est triée d'une certaine manière. C'est pourquoi nous allons commencer par parler du *classement complexe* des éléments.

À la fin du chapitre vous devrez être capable de :

- ▷ Détecter qu'on se trouve bien face à un problème qui peut entrer dans le cadre d'un algorithme de rupture et identifier si le tri des éléments est adéquat.
- ▷ Adapter le squelette général de l'algorithme de rupture au problème donné.

4.1 Le classement complexe

Introduction

Dans le chapitre sur les tris du cours d'algorithmique I (DEV1), vous avez abordé naturellement la notion du classement des données. Néanmoins, les données étaient « simples » : nombres ou chaînes, pour lesquelles la relation d'ordre est évidente. Les algorithmes mis en œuvre peuvent facilement s'adapter pour d'autres types, par exemple des objets `Date`, où l'opérateur de comparaison est remplacé par la méthode « `estAntérieure()` ».

Mais, plus généralement, les données composées de plusieurs champs (les structures) ou attributs (les objets) ne possèdent pas de relation d'ordre naturelle. C'est le cas, par exemple, des points d'un espace à deux ou trois dimensions ou encore des informations figurant sur une carte d'identité. Si on veut ordonner une série de telles données, il faudra choisir un premier critère de classement (par exemple le nom ou la date de naissance) et en cas d'égalité sur le premier critère (deux personnes peuvent avoir un même nom ou être nées le même jour), il faudra départager sur un second critère, et ainsi de suite.

Ces critères de classement sont bien entendu arbitraires, et dépendent de l'information qu'on veut retirer de l'ensemble des données. Notons aussi que l'ordre de classement peut être, pour chaque critère, croissant ou décroissant.

Exemple de classement simple

Prenons l'exemple d'une structure `Etudiant`, contenant les champs `matricule`, `nom`, `prénom`, `dateNaissance` et `option` (G, I ou R). Pour l'exemple, considérons une liste de 6 étudiants :

matricule	nom	prénom	dateNaissance	option
29845	Durant	Kevin	20/01/94	R
30125	Dupont	Fabrice	13/06/94	G
30351	Simon	André	18/11/94	G
30597	Dupont	Charles	9/07/94	G
31857	Guilmant	Léon	17/03/96	R
31886	Durant	Sam	30/05/94	I

Cette liste est classée sur le numéro de matricule. C'est un classement simple réalisé sur un seul champ des données. Le numéro de matricule étant dans ce cas-ci un **identifiant** des données, le problème de devoir départager ne se pose pas.

Exemple de classement double

Si nous désirons à présent classer sur l'ordre alphabétique des noms, il faut décider de départager, en cas de noms identiques, sur un autre champ, de façon naturelle sur celui des prénoms. Ceci donnerait le classement double suivant, en **majeur** sur le nom et en **mineur** sur le prénom :

matricule	nom	prénom	dateNaissance	option
30597	Dupont	Charles	9/07/94	G
30125	Dupont	Fabrice	13/06/94	G
29845	Durant	Kevin	20/01/94	R
31886	Durant	Sam	30/05/94	I
31857	Guilmant	Léon	17/03/96	R
30351	Simon	André	18/11/94	G

Exemple de classement triple

Supposons enfin que nous voulions grouper les étudiants par sections, nous devons alors classer prioritairement sur l'option, départager sur les noms et ensuite sur les prénoms. C'est alors un classement triple : en **majeur** sur l'option, en **médian** sur le nom et en **mineur** sur le prénom :

matricule	nom	prénom	dateNaissance	option
30597	Dupont	Charles	9/07/94	G
30125	Dupont	Fabrice	13/06/94	G
30351	Simon	André	18/11/94	G
31886	Durant	Sam	30/05/94	I
29845	Durant	Kevin	20/01/94	R
31857	Guilmant	Léon	17/03/96	R

Remarque : un classement n'est pas forcément un classement alphabétique. Par exemple, dans le cas du classement sur l'option, toute autre permutation des lettres G, I, R serait un tri possible.

Résumé

Les exemples ci-dessus constituent des exemples de **classements complexes**. On dira que des données sont classées sur la **clé composée** champ 1 – champ 2 – ... – champ i – ... – champ n (où « champ i » est un champ de la structure des données) si le classement se fait prioritairement depuis le champ 1 jusqu'au champ n . Autrement dit, si deux données ont tous leurs champs 1, 2, ..., i égaux ($i < n$), le classement se fait en départageant sur le champ $i + 1$. L'indice du champ correspond au **niveau** du classement complexe.

4.2 La notion de rupture

Les algorithmes que nous allons voir peuvent s'appliquer à n'importe quel **ensemble logique** d'éléments qui peut faire l'objet d'un traitement séquentiel (les listes, les tableaux...) ¹. Les éléments peuvent être de n'importe quel **type complexe** (variables structurées, objets...). Nos exemples seront souvent pris sur des listes de variables structurées.

Nous parlons de **rupture** lorsque, dans ce traitement itératif, on constate que l'information courante que l'on souhaite traiter n'appartient plus à l'ensemble (ou au sous-ensemble) des informations déjà traitées précédemment.

Lorsque les données sont triées selon une clé complexe, il est naturel de parler de *rupture sur un champ* de ces données, ou de rupture de niveau donné (voir section suivante).

Par exemple, dans le dernier classement des étudiants ci-dessus, il y a rupture sur l'option au niveau de Durant Sam et de Durant Kevin. En effet, ces deux étudiants délimitent les sous-ensembles d'étudiants partageant une même option.

Dans le 2^e classement, nous pouvons parler de rupture sur les noms : l'étudiant Durant Kevin met fin au sous-ensemble des Dupont, et l'étudiant Guilmant met fin à celui des Durant.

Dans le 1^{er} classement, qui est un classement simple sur le numéro de matricule, on peut considérer qu'il n'y a qu'un ensemble de données d'un seul tenant sans ruptures (ou avec rupture de niveau 0, voir section suivante), ou alors qu'il y a rupture à chaque étudiant, puisque chaque étudiant forme un sous-ensemble isolé par son numéro de matricule, vu qu'ils sont obligatoirement distincts (mais dans ce cas nous n'utilisons pas l'algorithme de rupture : c'est une simple itération sur la liste).

4.3 Traitement des ruptures dans une séquence ordonnée

Rupture de niveau 0

Quel que soit l'ordre de tri des données de l'ensemble parcouru séquentiellement, il est toujours possible de détecter la fin des données grâce à sa taille ². Cette « fin » constitue donc la rupture principale, celle signalant la fin du parcours itératif.

Sans le savoir, nous avons donc déjà traité la rupture générale d'un ensemble de données (c'est la rupture de « niveau 0 », car elle n'est pas liée à un champ des données, et est naturellement prioritaire sur ces champs). Pour illustrer cela, reprenons l'exemple de la liste d'étudiants. Le parcours de base de cet ensemble est le suivant ³ :

```

algorithm RuptureNiveau0(etudiants: List of Etudiant)
  i: integer
  i = 0
  while i < etudiants.taille()
    // traitement de etudiants.get(i)
    i = i + 1
  end
end

```

Si nous voulons faire des statistiques globales sur l'ensemble des étudiants (par ex. simplement les compter), le traitement de l'information consiste à incrémenter un compteur, et l'algorithme ci-dessus peut fonctionner quel que soit l'ordre de classement choisi.

1. Et également les fichiers moyennant une petite adaptation liée au fait qu'on ne connaît pas, dans ce cas, la taille de l'ensemble lorsqu'on commence à le traiter.

2. Ou via une marque spéciale de *fin de fichier* dans les cas des fichiers séquentiels.

3. Vous vous demandez peut-être pourquoi ne pas utiliser le « pour » ? Ce serait tout-à-fait valable dans ce cas précis mais ça ne pourra pas être généralisé lorsqu'il y a plusieurs niveaux de rupture.

Rupture de niveau 1

Passons à présent au « niveau 1 » ; c'est-à-dire un traitement de rupture correspondant à un classement complexe sur un champ. Imaginons que nous voulions savoir quel est le nombre d'étudiants dans chaque option. Une solution, sans algorithme de rupture, consisterait à avoir 3 compteurs, un par option. On peut imaginer une façon plus judicieuse de faire à partir du dernier classement, celui qui contient précisément les étudiants déjà groupés par option : à chaque fois qu'il y a rupture sur l'option, on peut alors connaître le total d'étudiants dans l'option qui vient d'être parcourue. Ceci ne nécessite qu'un seul compteur remis à 0 à chaque fois qu'une nouvelle option est rencontrée, c'est-à-dire à chaque rupture. De plus l'algorithme fonctionnera quel que soit le nombre d'options. Voici une telle solution :

```
algorithm RuptureNiveau1(etudiants: List of Etudiant)
    // on suppose les données classées en majeur sur l'option
    saveOption: string
    cpt: integer
    i: integer

    i = 0
    while i < etudiants.taille()
        saveOption = etudiants.get(i).option
        cpt = 0
        while i < etudiants.taille() ET etudiants.get(i).option = saveOption
            cpt = cpt + 1
            i = i + 1
        end
        print cpt, « étudiant dans l'option », saveOption
    end
end
```

Questions de réflexion :

- ▷ pourquoi la condition `i < etudiants.taille()` apparaît-elle une 2^e fois dans la boucle intérieure ?
- ▷ pourquoi est-ce que `i` et `cpt` ne sont pas initialisés au même endroit ?
- ▷ pourquoi l'incréméntation de `i` se fait-elle dans la boucle centrale et pas ailleurs ?
- ▷ pourquoi utilise-t-on `saveOption` plutôt que `etudiants.get(i).option` dans l'instruction d'affichage ?
- ▷ l'ordre des conditions apparaissant dans le 2^e « tant que » est-il important ?

Rupture de niveau 2

L'algorithme ci-dessus se généralise facilement si on ajoute davantage de niveaux de rupture. Pour illustrer le « niveau 2 », prenons encore l'exemple suivant : on veut connaître pour chaque option le nombre d'étudiants nés dans les différentes années de naissance. L'algorithme correspondant s'écrit facilement et fonctionne lorsque les données sont, cette fois-ci, classées en majeur sur l'option et en mineur sur la date de naissance (ou encore classement double sur la clé composée option – dateNaissance) :

```

algorithm RuptureNiveau2(etudiants: List of Etudiant)
  // on suppose les données classées en majeur sur l'option
  // et en mineur sur la date de naissance (ordre chronologique)
  saveOption: string
  saveAnnéeNaissance: integer
  cpt: integer
  i: integer
  i = 0
  while i < etudiants.taille()
    saveOption = etudiants.get(i).option
    while i < etudiants.taille() ET etudiants.get(i).option = saveOption
      saveAnnéeNaissance = etudiants.get(i).dateNaissance.année
      cpt = 0
      while i < etudiants.taille() ET etudiants.get(i).option = saveOption ET
        etudiants.get(i).dateNaissance.année = saveAnnéeNaissance
        cpt = cpt + 1
        i = i + 1
      end
      print cpt, " étudiant dans l'option ", saveOption, " sont nés en ", saveAnnéeNaissance
    end
    i = i + 1
  end

```

Ces exemples montrent que l'algorithme de rupture et le tri des listes sont étroitement liés. La structure de l'algorithme épouse le schéma de la clé composée du classement des données, et à un classement déterminé correspondra un algorithme bien précis.

4.4 Traitements de clôture et d'initialisation

Chaque rupture du traitement itératif des éléments d'un ensemble entraîne un **traitement de clôture** sur cet ensemble. Comme une rupture à un niveau implique des ruptures en cascade sur tous les niveaux d'ordres plus grands, un traitement de clôture d'un ensemble ne pourra se faire que lorsque le dernier sous-ensemble de cet ensemble sera clôturé.

De la même manière, l'arrivée d'un élément appartenant à un nouvel ensemble nécessite un **traitement d'initialisation** de ce nouvel ensemble.

En fait, il ne s'agit que de généraliser ce qui se fait au niveau 0 (travaux d'initialisation consistant par exemple à mettre des totalisateurs ou compteurs à zéro et travaux de clôture consistant par exemple à imprimer des résultats totaux particuliers) à tous les ensembles et sous-ensembles !

4.5 Exercices

Exercice 1

La chasse au gaspi [rupture de niveau 1]

À l'ÉSI, les quantités de feuilles imprimées et photocopiées par les professeurs et les étudiants sont enregistrées à des fins de traitement. Le service technique désirant facturer les « exagérations », vous fournit une liste de toutes les impressions effectuées depuis le début de l'année. Cette liste présente la structure d'enregistrement Job suivante et est ordonnée alphabétiquement **en majeur** sur le champ login :

```

structure Job
  login: string
  date: date
  nombre: integer
end

```

Écrire un algorithme permettant d'afficher une ligne par utilisateur dont le nombre total de feuilles imprimées dépasse une valeur limite entrée en paramètre. Cette ligne contiendra le login et le nombre.

Exercice 2 Compter le nombre d'étudiants par option

Reprenons l'exemple donné pour la rupture de niveau 2 (RuptureNiveau2, page 37). Que faut-il ajouter à l'algorithme pour qu'il affiche également le nombre total d'étudiants par option ?

Exercice 3 Compter les étudiants [rupture de niveau 2]

Supposons que la structure `Etudiant` contienne également un champ indiquant dans quel bloc se trouve l'étudiant (1, 2 ou 3). On voudrait un algorithme qui reçoit une liste d'étudiants et calcule le nombre d'étudiants dans chaque section et, par section, dans chaque bloc.

L'affichage ressemblera à :

```

Gestion
  bloc 1 : 130 étudiants
  bloc 2 : 42 étudiants
  bloc 3 : 16 étudiants
  TOTAL  : 188 étudiants
Industriel
  bloc 1 : 32 étudiants
  bloc 2 : 14 étudiants
  bloc 3 : 8 étudiants
  TOTAL  : 54 étudiants
Réseau
  bloc 1 : 82 étudiants
  bloc 2 : 31 étudiants
  bloc 3 : 13 étudiants
  TOTAL  : 126 étudiants

```

- Quel doit-être le tri de la liste pour pouvoir résoudre cet exercice avec un algorithme de rupture ?
- Écrire cet algorithme.

Exercice 4 Les fanas d'info [rupture de niveau 2]

Une grande société d'informatique a organisé durant les douze derniers mois une multitude de concours ouverts aux membres de clubs d'informatique. Elle souhaiterait récompenser le club qui aura été le plus « méritant » durant cette période au point de vue de la participation des membres mineurs. Chaque résultat individuel des participants (y compris des majeurs) est repris dans une liste dont les éléments sont de type `Participant`.

```

structure Participant
  nom: string
  âge: integer
  référence: string
  numéro: integer
  résultat: integer
end

```

// nom et prénom du participant
 // âge du participant au moment du concours
 // référence du club auquel appartient ce participant
 // numéro du concours auquel il a participé
 // résultat obtenu lors de ce concours (sur 100)

Sachant que la liste est ordonnée **en majeur sur le champ référence et en mineur sur le champ nom**, on demande d'écrire l'algorithme qui affiche les informations suivantes :

pour chaque club :

- ▷ sa référence
- ▷ pour chaque membre mineur de ce club :
 - ▷ son nom et prénom
 - ▷ la cote moyenne sur 100 des concours auquel ce membre a participé
- ▷ le nombre total de participations des membres mineurs

N.B. : un membre mineur qui s'est inscrit à un concours = une participation. Un club qui n'aura eu aucun membre mineur participant figurera quand même dans le résultat avec la mention « Pas de participation de membre mineur ». Par contre, un club dont aucun membre n'a participé au moindre concours ne sera pas affiché.

À la fin, on affichera la référence du meilleur club, à savoir celui qui a eu la plus haute cote moyenne de membres mineurs (simplifions on ne gérant pas les possibles ex-æquo).

Exercice 5

Éliminer les doublons d'une liste.

Soit une liste ordonnée d'entiers avec des possibles redondances. Écrire un algorithme qui enlève les redondances de la liste. On vous demande de créer une nouvelle liste (la liste de départ reste inchangée).

Exemple : si la liste est (1, 3, 3, 7, 8, 8, 8) le résultat sera (1, 3, 7, 8).

Exercice 6

Une suite logique

Voici une petite suite logique :

```
1
1 1
2 1
1 2 1 1
1 1 1 2 2 1
3 1 2 2 1 1
1 3 1 1 2 2 2 1
1 1 1 3 2 1 3 2 1 1
3 1 1 3 1 2 1 1 1 3 1 2 2 1
...
```

- a) Comprendre la logique derrière cette suite et écrire la ligne suivante.
- b) Écrire un algorithme qui reçoit une ligne (sous forme d'une liste d'entiers) et retourne la ligne suivante (sous forme d'une autre liste d'entiers). Votre première tâche sera probablement de comprendre ce que vient faire cet exercice dans le chapitre des ruptures puisque la liste n'est pas triée.
- c) Écrire l'algorithme qui reçoit N (un entier) et affiche les N premières lignes de cette suite logique.

Exercice 7

Alternative à la rupture

Reprenons l'exemple donné pour la rupture de niveau 1 ([RuptureNiveau1](#), page 36). Supposons que la liste ne soit **pas** triée sur l'option. Écrivez l'algorithme qui calcule le nombre d'étudiants par option en un seul parcours de la liste (vous devrez utiliser trois compteurs distincts).

Représentation des données

Nous voici arrivés au terme de ce cours d'algorithmique. Ce chapitre apporte une synthèse des différentes notions vues tout au long de vos cours d'algorithmiques de 1^{re} année et propose quelques pistes de réflexion quant au choix d'une bonne représentation des données qui se pose lors de la résolution de problèmes de programmation avancés.

Pour la plupart de ces exercices, la difficulté tient en partie dans le bon choix d'une représentation des données et de la démarche algorithmique la plus adéquate à mettre en œuvre pour agir sur ces données en vue d'obtenir le résultat escompté. Noter que l'efficacité d'un algorithme est liée étroitement au choix de la représentation.

5.1 Se poser les bonnes questions

Revenons à la case départ : nous avons commencé le cours d'algorithmique de DEV1 en situant les notions de **problème** et de **résolution**. Nous avons vu qu'un problème bien spécifié s'inscrit dans le schéma :

étant donné [la situation de départ] on demande [l'objectif]

Une fois le problème correctement posé, on peut partir à la recherche d'une **méthode de résolution**, c'est-à-dire d'un algorithme en ce qui concerne les problèmes à résoudre par les moyens informatiques.

Tout au long de l'année, nous avons vu divers modèles et techniques algorithmiques adaptés à des structures particulières (les nombres, les chaînes, les tableaux, les variables structurées, les objets, les listes...). La plupart des exercices portaient directement sur ces structures (par ex. calculer la somme des nombres d'un tableau, extraire une sous-liste à partir d'une liste donnée). Ces exercices d'entraînement et de formation quelque peu théoriques constituent en fait des démarches algorithmiques de base qui trouvent toutes une place dans des problèmes plus complexes.

Mais la plupart des problèmes issus des situations de la vie courante auxquels se confronte le programmeur s'expriment généralement de manière plus floue : par ex. dresser la comptabilité des dépenses mensuelles d'une firme, faire un tableau récapitulatif du résultat des élections par cantons électoraux, faire une version informatique d'un jeu télévisé... Les exemples sont infinis !

C'est dans le cadre de ce genre de problème plus complexe que se pose le problème de la **représentation de données**. Une fois le problème bien spécifié (par les données et l'objectif) apparaissent naturellement les questions suivantes : quelles données du problème sont réellement utiles à sa résolution ? (Il est fréquent que l'énoncé d'un problème contienne des données superflues ou inutiles). Y a-t-il des données plus importantes que d'autres ? (données principales ou secondaires). Les données doivent-elles être consultées plusieurs fois ? Quelles données faut-il conserver en mémoire ? Sous quelle forme ? Faut-il utiliser un tableau ? Une liste ? Faut-il créer une nouvelle classe ? Les données doivent-elles être classées suivant un critère précis ? Ou la présentation brute des données suffit-elle pour solutionner le problème posé ?

Les réponses ne sont pas directes, et les différents outils qui sont à notre disposition peuvent être ou ne pas être utilisés. Il n'y a pas de règles précises pour répondre à ces questions, c'est le flair et le savoir-faire développés patiemment par le programmeur au fil de ses expériences et de son apprentissage qui le guideront vers la solution la plus efficace. Parfois plusieurs solutions peuvent fonctionner sans pouvoir départager la meilleure d'entre elles.

Ce type de questionnement est peut-être l'aspect le plus délicat et le plus difficile de l'activité de programmation, car d'une réponse appropriée dépendra toute l'efficacité du code développé. Un mauvais choix de représentation des données peut mener à un code lourd et maladroit. En vous accompagnant dans la résolution des exercices qui suivent, nous vous donnerons quelques indices et pistes de réflexion, qui seront consolidées par l'expérience acquise lors des laboratoires de langages informatiques ainsi que par les techniques de modélisation vues au cours d'analyse.

5.2 Les structures de données

Rappelons brièvement les différentes structures étudiées dans ce cours :

- ▷ les **données « simples »** (variables isolées : entiers, réels, chaînes, caractères, booléens) ;
- ▷ les **variables structurées**, qui regroupent en une seule entité une collection de variables simples ;
- ▷ le **tableau**, qui contient un nombre déterminé de variables de même type, accessibles via un indice ou plusieurs pour les tableaux multidimensionnels ;
- ▷ les **objets**, qui combinent en un tout une série d'attributs et des méthodes agissant sur ces attributs ;
- ▷ la **liste**, qui peut contenir un nombre indéfini d'éléments de même type.

D'autres structures particulières s'ajouteront dans le cours d'algorithmique d'ALG3 : les listes chaînées, les piles, les files, les arbres, les associations et les graphes.

Chacune de ces structures possède ses spécificités propres quant à la façon d'accéder aux valeurs, de les parcourir, de les modifier, d'ajouter ou de supprimer des éléments à la collection.

5.3 Exercices

Exercice 1 La course à la case 64 à 4 joueurs

Commençons par un petit jeu très simple de course avec un dé, dont voici les règles.

« Ce jeu se joue à 4 joueurs qui doivent parcourir un chemin de 64 cases. Ils commencent tous sur la case 1 et jouent à tour de rôle (en commençant par le premier joueur). À son tour, le joueur lance un dé à 6 faces et avance du nombre de cases indiqué par le dé. Le premier joueur à atteindre ou dépasser la case 64 a gagné. Seule contrainte, un joueur ne peut pas terminer son tour sur une case occupée. Si c'est le cas, il avance jusqu'à la case libre suivante. »

Voici 3 propositions de représentation de données. On vous demande pour chaque proposition de vérifier, sans écrire l'algorithme, si elle permet la programmation du jeu. On vous conseille vivement de « dessiner »¹ les propositions pour mieux les comprendre.

1. Un tableau de 64 entiers. La case k contient i si le joueur i s'y trouve ou 0 si la case est libre. Mais aussi un entier `joueurCourant` donnant le numéro du joueur courant.

1. Par là, on veut dire : imaginer une situation de jeu (positions des joueurs sur le chemin par exemple) et voir quelles valeurs doivent avoir les variables introduites dans la représentation pour correspondre à cette situation de jeu.

2. Un tableau de 4 entiers. La case i contient la position du joueur i . Mais aussi un entier `joueurCourant` donnant le numéro du joueur courant.
3. On combine les deux premières propositions (on a donc deux tableaux).

Après ces vérifications vous choisirez une des représentations pour écrire la solution (non OO à ce stade) du jeu. Pensez à découper votre solution.

Exercice 2 La course à la case 64 à n joueurs

Modifiez l'exercice précédent afin que le jeu puisse se jouer à n joueurs, où n est un entier supérieur ou égal à 2, choisi au début du jeu.

Exercice 3 La course à la case 64 à n joueurs - variantes

Reprenons la course à la case 64 de l'exercice précédent. Voici quelques propositions de modification des règles. Pour chaque proposition, indiquez si la représentation choisie dans l'exercice précédent est toujours valable et pertinente.

1. Si un joueur arrive sur une case occupée, le joueur qui s'y trouvait retourne à la première case.
2. Si un joueur termine sa course sur une case qui est un multiple de 5, il rejoue directement.
3. Un joueur rejoue directement s'il termine sa course sur les cases 1, 2, 7, 11, 17, 31, 42 ou 53.

Exercice 4 Un jeu de poursuite

Deux joueurs A et B se poursuivent sur un circuit de 50 cases. Au départ, A se trouve sur la case 1 et B est placé sur la case 26. C'est A qui commence. Chaque joueur joue à son tour en lançant un dé dont la valeur donne le nombre de cases duquel il doit avancer sur le jeu. Lorsqu'un joueur arrive sur la case 50 et qu'il doit encore avancer, il continue son parcours à partir de la case 1. Le jeu se termine lorsqu'un joueur rattrape ou dépasse l'autre.

Écrire un algorithme (non OO pour le moment) de simulation de ce jeu qui se terminera par l'affichage du vainqueur ainsi que le nombre de tours complets parcourus par ce vainqueur.

Il est important de bien découper votre algorithme. On vous suggère d'écrire les algorithmes suivant :

- ▷ un algorithme `initialiser()` qui initialise le jeu (placement des joueurs...);
- ▷ un algorithme « `jouerCoup` » qui joue pour un joueur et indique s'il a rattrapé l'autre joueur;
- ▷ un algorithme « `joueurSuivant` » qui permet de passer au joueur suivant.

À nouveau, on vous fait plusieurs propositions pour la représentation de l'état du jeu. On vous demande pour chacune d'elles de vérifier, sans écrire les méthodes de la classe, si elles permettent la programmation du jeu. Après ces vérifications vous choisirez une des représentations pour écrire la classe complète.

1. Dans cette proposition, nous avons deux variables.
 - ▷ `circuit` : un tableau de 1 à 50 chaînes de caractères. Les chaînes de caractères représenteront la position des joueurs (au départ, "A" en 1 et "B" en 26, " " dans les autres positions).
 - ▷ `joueurCourant` : un entier donnant la position du joueur courant.
2. Cette proposition introduit une structure `Joueur` et le nombre de tours.
 - ▷ `circuit` : un tableau de 1 à 50 éléments `Joueur`, une structure.

```

structure Joueur
|   nom: string           // Le nom du joueur à cette position ("A", "B" ou " " si la case est vide)
|   nbTours: integer      // Nb de tours qu'a fait le joueur qui est à cette position (0 si case vide)
end
```

Un joueur a fait un tour complet quand il est de nouveau sur sa position de départ ou la dépasse.

▷ `joueurCourant` : un entier donnant la position du joueur courant.

3. Dans cette proposition, le tableau change de signification.

▷ `circuit` : un tableau de 2 éléments `Joueur`, une structure différente.

```
structure Joueur
| position: integer    // Donne la position du joueur sur le circuit (entier entre 1 et 50)
| nbTours: integer     // Le nombre de tours qu'a fait ce joueur
end
```

▷ `joueurCourant` : un entier donnant la position du joueur courant.

4. Cette proposition est identique à la précédente sauf sur un point :

▷ On ne retient plus le nombre de tours effectués mais simplement le nombre de cases parcourues. Par exemple, si un joueur a fait exactement deux tours complets, le nombre de cases parcourues sera de 100.

```
structure Joueur
| position: integer    // Donne la position du joueur sur le circuit (entier entre 1 et 50)
| nbCasesParcourues: integer // Nb de cases parcourues par le joueur depuis le départ
end
```

Exercice 5

Un jeu de poursuite - variante

Dans cette variante, chaque case contient une valeur vrai ou faux indiquant si le joueur pourra rejouer. Si la case sur laquelle tombe le joueur contient la valeur `vrai` il avance encore une fois du même nombre de cases (et de même s'il tombe encore sur `vrai`).

Qu'est-ce que cela change au niveau des données ? Modifiez votre solution en conséquence.

Pour adapter le code, il faudra adapter les paramètres fournis à l'algorithme d'initialisation et à l'algorithme qui joue un coup. Nous vous conseillons également de ne pas modifier l'algorithme `jouerCoup` mais d'introduire un nouvel algorithme (par ex. `jouerTour`) qui y fait appel plusieurs fois si nécessaire.

Exercice 6

Le Jeu du Millionnaire

Un questionnaire de quinze questions à choix multiples de difficulté croissante est soumis à un candidat. Quatre possibilités de réponses (dont une seule est correcte) sont proposées à chaque fois. Au plus le candidat avance dans les bonnes réponses, au plus son gain est grand. S'il répond correctement aux quinze questions, il empoche la somme rondelette de 500.000 €.

Par contre, si le candidat donne une mauvaise réponse, il risque de perdre une partie du gain déjà acquis. Cependant, certains montants intermédiaires constituent des paliers, c'est-à-dire une somme acquise que le candidat est sûr d'empocher, quoiqu'il arrive dans la suite du jeu.

À chaque question, le candidat a donc trois possibilités :

- ▷ il donne la réponse correcte : dans ce cas il augmente son gain, et peut passer à la question suivante
- ▷ il ne connaît pas la réponse, et choisit de s'abstenir : dans ce cas, le jeu s'arrête et le candidat empoche le gain acquis à la question précédente
- ▷ il donne une réponse incorrecte : le jeu s'arrête également, mais le candidat ne recevra que le montant du dernier palier qu'il a atteint et réussi lors de son parcours. En particulier, si le candidat se trompe avant d'avoir atteint le premier palier, il ne gagne pas un seul euro !

1	25 €	faux
2	50 €	faux
3	125 €	faux
4	250 €	faux
5	500 €	vrai
6	1000 €	faux
7	2000 €	faux
8	3750 €	faux
9	7500 €	faux
10	12500 €	vrai
11	25000 €	faux
12	50000 €	faux
13	100000 €	vrai
14	250000 €	faux
15	500000 €	vrai

Exemple : Le tableau ci-contre contient les gains associés à chaque question et une indication booléenne mise à **vrai** lorsque la question constitue un palier. Un concurrent qui se trompe à la question 3 ne gagnera rien ; un concurrent qui se trompe à la question 6 gagnera 500 € (palier de la question 5) et de même s'il se trompe à la question 10 ; un concurrent qui se trompe à la question 13 gagnera 12500 € (palier de la question 10) ; s'il décide de ne pas répondre à la question 13, il garde le montant acquis à la question 12, soit 50000 €.

Il y aurait de nombreuses façons de coder ce problème ; en voici une :

La structure Question

Une question est composée du libellé de la question, des 4 libellés pour les réponses et d'une indication de la bonne réponse (un entier de 1 à 4). Par simplicité on en fait une structure mais on pourrait en faire une classe si on voulait par exemple vérifier que la « bonne réponse » possède une valeur correcte.

La structure Gain

Représente un niveau de gain. Elle contient les champs : montant (entier) et palier (un booléen à **vrai** si cette somme est assurée, **faux** sinon)

La classe Millionnaire

Cette classe code le moteur du jeu. On y retrouve

- ▷ questionnaire : un tableau de Question
- ▷ gains : un tableau de Gain
- ▷ autres attributs à déterminer (cf. méthodes)

ainsi que les méthodes pour

- ▷ initialiser le jeu à partir d'un questionnaire et du tableau de gains
- ▷ connaître la question en cours
- ▷ donner la réponse du candidat à la question en cours
- ▷ savoir si le jeu est fini ou pas
- ▷ arrêter le jeu en repartant avec les gains
- ▷ les accesseurs nécessaires pour connaître l'état du jeu.

Le jeu proprement dit

L'algorithme `jeuMillionnaireConsole()` reçoit le questionnaire et les gains et simule le jeu :

- ▷ Il propose les questions au candidat
- ▷ Il lit ses réponses (chiffre 1 à 4 ou 0 pour arrêter) et fait évoluer le jeu en fonction.
- ▷ lorsque le jeu est terminé, il indique au candidat le montant de ses gains.
- ▷ Attention ! Cet algorithme devrait être le plus petit possible. Imaginez que vous devez également coder une version graphique. Tout code commun doit se trouver dans la classe Millionnaire !

Exercice 7

Chambre avec vue

Un grand hôtel a décidé d'informatiser sa gestion administrative. Il a confié ce travail à la société ESI_INFO dans laquelle vous êtes un informaticien chevronné. On vous a confié la

tâche particulière de la gestion des réservations pour ses 100 chambres. Pour ce faire, on vous demande d'écrire une classe **Hôtel** qui offre notamment une méthode qui permet d'enregistrer une réservation.

Pour représenter l'occupation des chambres un jour donné, nous allons utiliser un tableau de 100 entiers. Un 0 indique que la chambre est libre, une autre valeur (positive) indique le numéro du client qui occupe cette chambre ce jour-là.

Nous utiliserons une Liste de tels tableaux pour représenter l'occupation des chambres sur une longue période ; les éléments se suivant correspondant à des jours successifs.

Nous vous imposons les attributs de la classe, à savoir :

- ▷ **occupations** : une Liste de tableaux de 100 entiers comme expliqué ci-dessus.
- ▷ **premierJour** : donne le jour concerné par le premier élément de la liste. Ainsi s'il vaut 10/9/2015 cela signifie que le premier élément de la liste « occupations » renseigne sur l'occupation des chambres ce 10/9/2015 ; que le deuxième élément de la liste concerne le 11/9/2015 et ainsi de suite...

Écrire la méthode suivante

method effectuerRéservation(demande ↓ : DemandeRéservation, chambre ↑ : integer) → boolean

où la structure de demande de réservation est définie ainsi

```
structure DemandeRéservation
|   numéroClient: integer
|   débutRéservation: Date
|   nbNuitées: integer
end
```

- ▷ Le booléen retourné indique si la réservation a pu se faire ou pas
- ▷ Si elle a pu se faire, le paramètre de sortie **chambre** indique la chambre qui a été choisie
- ▷ Si plusieurs chambres sont libres, on choisit celle avec le plus petit numéro
- ▷ La demande de réservation peut couvrir une période qui n'est pas encore reprise dans la liste ; il faudra alors l'agrandir

Exercice 8

L'ensemble

La notion d'ensemble fini est une notion qui vous est déjà familière pour l'avoir rencontrée dans plusieurs cours. Nous rappelons certaines de ses propriétés et opérations.

Étant donnés deux ensembles finis **S** et **T** ainsi qu'un élément **x** :

- ▷ $x \in S$ signifie que l'élément **x** est un élément de l'ensemble **S**.
- ▷ L'ensemble vide, noté \emptyset est l'ensemble qui n'a pas d'élément ($x \in \emptyset$ est faux quel que soit **x**).
- ▷ L'ordre des éléments dans un ensemble n'a aucune signification, l'ensemble $\{1,2\}$ est identique à $\{2,1\}$.
- ▷ Un élément **x** ne peut pas être plus d'une fois élément d'un même ensemble (pas de répétition).
- ▷ L'union $S \cup T$ est l'ensemble contenant les éléments qui sont dans **S** ou (non exclusif) dans **T**.
- ▷ L'intersection $S \cap T$ est l'ensemble des éléments qui sont à la fois dans **S** et dans **T**.
- ▷ La différence $S \setminus T$ est l'ensemble des éléments qui sont dans **S** mais pas dans **T**.

Créer la classe `Ensemble` décrite ci-dessous (où `E` est le type des éléments de l'ensemble).

```
class Ensemble de E
public :
    constructor Ensemble de E           // construit un ensemble vide
    method ajouter(elt : E)             // ajoute l'élément à l'ensemble
    method enlever(elt : E)             // enlève un élément de l'ensemble
    method contient(elt : E) → boolean  // dit si l'élément est présent
    method estVide() → boolean          // dit si l'ensemble est vide
    method taille() → integer           // donne la taille de l'ensemble
    method union(autreEnsemble : Ensemble de E) → Ensemble de E
    method intersection(autreEnsemble : Ensemble de E) → Ensemble de E
    method moins(autreEnsemble : Ensemble de E) → Ensemble de E
    method éléments() → List of E      // conversion en liste
end
```

Quelques remarques :

- ▷ La méthode d'ajout (resp. de suppression) n'a pas d'effet si l'élément est déjà (resp. n'est pas) dans l'ensemble.
- ▷ Les méthodes `union()`, `intersection()` et `moins()` retournent un troisième ensemble, résultat des 2 premiers sans toucher à ces 2 ensembles. On aurait pu envisager des méthodes modifiant l'ensemble sur lequel on les appelle.
- ▷ La méthode `éléments()` est nécessaire si on veut parcourir les éléments de l'ensemble (par exemple pour les afficher).

Exercice 9

La course à la case 64 en version OO

Reprenez la solution que vous avez écrite pour l'exercice 2 (La course à la case 64 à n joueurs) et voyez comment le transformer pour en faire une version OO.

Exercice 10

Un jeu de poursuite en version OO

Reprenez la solution que vous avez écrite pour l'exercice 5 (Un jeu de poursuite - variante) et voyez comment le transformer pour en faire une version OO.

Exercice 11

Les congés

Les périodes de congés des différents employés d'une firme sont reprises dans un tableau booléen **Congés** bidimensionnel à n lignes et 366 colonnes. Chaque ligne du tableau correspond à un employé et chaque colonne à un jour de l'année. Une case est mise à **vrai** si l'employé correspondant est en congé le jour correspondant. La firme en question est opérationnelle 7 jours sur 7, on n'y fait donc pas de distinction entre jours ouvrables, week-end et jours fériés.

Ce tableau permet de visualiser l'ensemble des congés des travailleurs, et d'accorder ou non une demande de congé, suivant les règles suivantes :

1. une période de congé ne peut excéder 15 jours ;
2. un employé a droit à maximum 40 jours de congé par an ;
3. à tout moment, 50% des employés doivent être présents dans la firme.

Écrire un algorithme qui détermine si cette demande peut être accordée ou non à un employé dont on connaît le nom, ainsi que les dates de début et de fin d'une demande de congé (objets de la classe `Date`). Dans l'affirmative, le tableau **Congés** sera mis à jour.

Pour établir la correspondance entre ce tableau et les noms des employés, vous avez à votre disposition un tableau **Personnel** de chaînes. L'emplacement du nom d'un employé dans ce tableau correspond à l'indice ligne du tableau **Congés**.

Il est permis d'utiliser pour résoudre cet exercice la méthode suivante de la classe `Date`, sans devoir détailler son code :

```
method numéroJour() → integer           // la position du jour dans l'année (entre 1 et 366)
```

Exercice 12**Casino**

Pour cet exercice, on vous demande un petit programme qui simule un jeu de roulette très simplifié dans un casino.

Dans ce jeu simplifié, vous pourrez miser une certaine somme et gagner ou perdre de l'argent (telle est la fortune, au casino !). Quand vous n'avez plus d'argent, vous avez perdu.

Notre règle du jeu

Bon, la roulette, c'est très sympathique comme jeu, mais un peu trop compliqué pour un exercice de première année. Alors, on va simplifier les règles et je vous présente tout de suite ce que l'on obtient :

- ▷ Le joueur mise sur un numéro compris entre 0 et 49 (50 numéros en tout). En choisissant son numéro, il y dépose la somme qu'il souhaite miser.
- ▷ La roulette est constituée de 50 cases allant naturellement de 0 à 49. Les numéros pairs sont de couleur noire, les numéros impairs sont de couleur rouge. Le croupier lance la roulette, lâche la bille et quand la roulette s'arrête, relève le numéro de la case dans laquelle la bille s'est arrêtée. Dans notre programme, nous ne reprendrons pas tous ces détails « matériels » mais ces explications sont aussi à l'intention de ceux qui ont eu la chance d'éviter les salles de casino jusqu'ici. Le numéro sur lequel s'est arrêtée la bille est, naturellement, le numéro gagnant.
- ▷ Si le numéro gagnant est celui sur lequel le joueur a misé (probabilité de $1/50$, plutôt faible), le croupier lui remet 3 fois la somme mise.
- ▷ Sinon, le croupier regarde si le numéro misé par le joueur est de la même couleur que le numéro gagnant (s'ils sont tous les deux pairs ou tous les deux impairs). Si c'est le cas, le croupier lui remet 50% de la somme mise. Si ce n'est pas le cas, le joueur perd sa mise.

Dans les deux scénarios gagnants vus ci-dessus (le numéro misé et le numéro gagnant sont identiques ou ont la même couleur), le croupier remet au joueur la somme initialement mise avant d'y ajouter ses gains. Cela veut dire que, dans ces deux scénarios, le joueur récupère de l'argent. Il n'y a que dans le troisième cas qu'il perd la somme mise.

Comme vous pouvez le constater, on ne vous fait pas de proposition pour la représentation des données. À vous de jouer !

Exercice 13**Mots croisés**

Voici une grille de mots croisés. (on ne s'intéresse pas ici aux définitions). Écrire une classe Grille offrant les méthodes suivantes :

- ▷ placer une lettre à un endroit de la grille (une case non noire bien sûr) ;
- ▷ donner le nombre de cases noires sur la grille ;
- ▷ donner le nombre total de mots (plus d'une lettre) de la grille (donc y compris ceux que le joueur n'a pas encore complétés) ;
- ▷ donner le nombre de mots déjà complétés par le joueur.

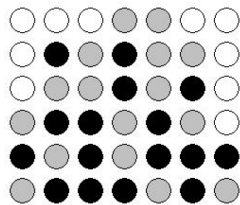
Exemple : dans la grille ci-contre, le nombre de cases noires est 14, le nombre total de mots de la grille est 37 (19 horizontaux et 18 verticaux) et le nombre de mots déjà complétés par le joueur est 6.

		A							
		L							
L	O	G	I	Q	U	E			
		O							
		R							
E	S	I		O		H			
		T	A	B	L	E	A	U	
		H		J		B			
		M		E					
		E		T					

Exercice 14

Puissance 4

Le jeu de puissance 4 se déroule dans un tableau vertical comportant 6 rangées et 7 colonnes dans lequel deux joueurs introduisent tour à tour des jetons (rouges pour l'un, jaunes pour l'autre). Avec l'aide de la gravité, les jetons tombent toujours le plus bas possible dans les colonnes où on les place. Le jeu s'achève lorsqu'un des joueurs a réussi à aligner 4 de ses jetons horizontalement, verticalement ou en oblique, ou lorsque les deux joueurs ont disposé chacun leur 21 jetons sans réaliser d'alignement (match nul).



N.B. : sur ce dessin noir et blanc, les jetons rouges apparaissent en noir, les jetons jaunes en gris et les cases blanches désignent l'absence de jetons. Cet exemple montre une situation du jeu où le joueur « jaune » est gagnant. En introduisant un jeton dans la 4^e colonne, il a réalisé un alignement de 4 jetons en oblique.

On demande d'implémenter une classe `Puissance4` qui permette de contrôler l'état des différentes phases du jeu. Déterminez les attributs de cette classe et décrivez-les brièvement de manière à justifier votre choix. Dotez ensuite la classe des méthodes permettant de :

- ▷ savoir si la grille est pleine
- ▷ mettre la grille à jour lorsque le joueur n (1 ou 2) joue dans la colonne j (entre 1 et 7). Cette méthode renverra la valeur booléenne faux si la colonne en question est déjà pleine
- ▷ vérifier si le joueur qui vient de jouer dans la colonne j a gagné la partie

N.B. : pour la structure qui contiendra le contenu du tableau de jetons, on adoptera la convention suivante : 0 pour l'absence de jeton, 1 représentera un jeton du 1^{er} joueur, et 2 un jeton du 2^e joueur (on peut donc faire abstraction de la couleur du jeton dans ce problème).

Exercice 15

Mastermind

Revenons sur le jeu Mastermind déjà vu en DEV1. Dans ce jeu, un joueur A doit trouver une combinaison de k pions de couleur, choisie et tenue secrète par un autre joueur B. Cette combinaison peut contenir éventuellement des pions de même couleur. À chaque proposition du joueur A, le joueur B indique le nombre de pions de la proposition qui sont corrects et bien placés et le nombre de pions corrects mais mal placés.

Exemple

Utilisons des lettres pour représenter les couleurs.

Combinaison secrète				
R	R	V	B	J

Proposition du joueur				
R	V	B	B	V

Il sera indiqué au joueur qu'il a :

- ▷ 2 pions bien placés : le R en 1^{re} position et le second B en 4^e position ;
- ▷ 1 pion mal placé : un des deux V (ils ne peuvent compter tous les deux).

Supposons une énumération `Couleur` (cf. la description d'une énumération en annexe) avec toutes les couleurs possibles de pion.

- a) Écrire une classe « `Combinaison` » pour représenter une combinaison de k pions. Elle possède une méthode pour générer une combinaison aléatoire (que vous ne devez pas écrire) et une méthode pour comparer une combinaison à la combinaison secrète (que vous devez écrire)
- b) Écrire ensuite une classe « `MasterMind` » qui représente le jeu et permet d'y jouer. La taille de la combinaison et le nombre d'essais permis seront des paramètres du constructeur.

Nous présentons ici quelques éléments qui pourront vous être utiles pour résoudre certains exercices.

A.1 Énumération

Parfois, une variable ne peut prendre qu'un ensemble fixe et fini de valeurs. Par exemple une variable représentant une saison ne peut prendre que quatre valeurs (HIVER, PRINTEMPS, ÉTÉ, AUTOMNE). On va l'indiquer grâce à l'énumération qui introduit un **nouveau type** de donnée.

```
enum Saison { HIVER, PRINTEMPS, ÉTÉ, AUTOMNE }
```

Il y a deux avantages à cela : une indication claire des possibilités de la variable lors de la déclaration et une lisibilité du code grâce à l'utilisation des valeurs explicites.

Par exemple,

```
// Lit une saison et affiche sa particularité
algorithm particularitéSaisonnière
| uneSaison: Saison
| read uneSaison           // on lira la valeur HIVER ou PRINTEMPS ou ÉTÉ ou AUTOMNE
| if uneSaison = HIVER
| | print "il neige"
| else if uneSaison = PRINTEMPS
| | print "les fleurs poussent"
| else if uneSaison = ÉTÉ
| | print "le soleil brille"
| else
| | print "les feuilles tombent"
| end
end
```

Exercice 1

Autres situations

Pouvez-vous identifier d'autres données qui pourraient avantageusement s'exprimer avec une énumération ?

A.1.1 Quid des langages de programmation ?

Certains langages (comme Java) proposent un type énuméré complet. D'autres (comme C et C++) proposent un type énuméré incomplet mais qui permet néanmoins une écriture comme celle ci-dessus. Cobol propose des « noms conditions » qui représentent l'ensemble des valeurs possibles d'une variable. D'autres langages, enfin, ne proposent rien. Pour ces langages, le truc

est de définir des constantes entières qui vont permettre une écriture proche de celle ci-dessus (mais sans une déclaration explicite).

A.1.2 Lien avec les entiers

Dans l'exemple ci-dessus, on lit une Saison mais souvent, si on travaille avec les Mois par exemple, on disposera plutôt d'un entier. Il faut pouvoir convertir les valeurs. Chaque langage de programmation propose sa propre technique ; nous allons adopter la syntaxe suivante :

```
Saison(3)                                // donne l'énumération de la saison numéro 3 (on commence à 1);
                                         // donne ÉTÉ dans notre exemple.
position(uneSaison)                     // donne l'entier associé à une saison ;
                                         // si on a lu HIVER comme valeur pour uneSaison, donne la valeur 1.
```

A.2 Gestion des erreurs

Lorsqu'un algorithme se trouve dans un état incorrect, particulièrement lorsqu'un paramètre est invalide, on peut l'indiquer via la primitive **error**.

Par exemple :

```
algorithm racineCarrée(nb: entier)
|   if nb<0
|   |   error "Le nombre doit être positif"
|   end
|   suite de l'algorithme
end
```

Pratiquement, cette primitive stoppe l'algorithme sans aucune possibilité de récupération. Dans un langage comme JAVA vous utiliserez le mécanisme des exceptions qui est plus souple.

Cet aide-mémoire peut vous accompagner lors d'une interrogation ou d'un examen. Il vous est permis d'utiliser ces méthodes sans les développer. Par contre, si vous sentez le besoin d'utiliser une méthode qui n'apparaît pas ici, il faudra en écrire explicitement le contenu.

Manipuler les nombres

```
method hasard() → réel // Donne un nombre réel entre 0 inclus et 1 exclu.
method hasard(min: entier, max: entier) → entier // Donne un entier entre min et max inclus.
```

La liste

```
// T est un type quelconque
class List of T
  public :
    constructor List of T // construit une liste vide
    method get(pos : integer) → T // donne un élément en position pos
    method set(pos : integer, valeur : T) // modifie un élément en position pos
    method taille() → integer // donne le nombre actuel d'éléments
    method ajouter(valeur : T) // ajoute un élément en fin de liste
    method insérer(pos : integer, valeur : T) // insère un élément en position pos
    method supprimer // supprime le dernier élément
    method supprimerPos(pos : integer) // supprime l'élément en position pos
    method supprimer(valeur : T) → boolean // supprime l'élément de valeur donnée
    method vider // vide la liste
    method estVide() → boolean // la liste est-elle vide ?
    method existe(valeur ↓ : T, pos ↑ : entier) → boolean // recherche un élément
end
```