

# Une introduction à l'algorithmique

Renée De Graeve

1<sup>er</sup> octobre 2009



# Introduction

Qu'est-ce qu'un algorithme ?

Un algorithme est la description d'une méthode pour effectuer une tâche précise : l'algorithme prend en compte des données de départ et donne les instructions à effectuer pour parvenir au résultat cherché.

La façon dont les instructions sont données importe peu, mais il est important que ces instructions ne soient pas ambiguës et soient valables pour toutes les données de départ.

Quel langage choisir ?

Cela dépend à qui l'algorithme est destiné :

- Si c'est à Monsieur "Toutlemonde" on utilisera sa langue habituelle,
- Si c'est à un ordinateur on utilisera un langage de programmation particulier pour cette machine.

Ici, on suppose que l'on va traduire les algorithmes en des langages de même type qui seront compréhensibles par différentes machines.

On choisira le langage du logiciel `Xcas` pour traduire les algorithmes et ainsi de pouvoir les faire fonctionner.



# Chapitre 1

## Jeu sur la notion d'algorithme

### 1.1 Description

On forme des petits groupes d'élèves de taille  $p$ . Chaque groupe doit mettre au point une stratégie pour gagner au jeu suivant :

- Chaque groupe passe à tour de rôle devant le meneur de jeu.
- Le meneur de jeu distribue à chaque élève 10 cartons de numéro  $0, 1, \dots, 9$ .
- Le meneur de jeu annonce un nombre  $n \leq 9p$  et les élèves du groupe ne doivent plus se concerter mais doivent appliquer leur stratégie en montrant un carton de façon à ce que le total des cartons fasse  $n$ .
- On répète cela 10 fois de suite pour chaque groupe. Pendant le jeu, les autres élèves observent et notent : le nombre annoncé et les réponses. Ils peuvent ainsi attribuer une note entre 0 et 10 au groupe.

### 1.2 Le but

Lorsque tous les groupes ont joué, on demande à chaque groupe de décrire leur stratégie : on peut voir alors, si cette stratégie répond au problème, si elle a été appliquée correctement par tous les membres du groupe, si elle est compréhensible, si il y a des ambiguïtés, si elle est valable dans tous les cas...

Le but de cette mise en commun est d'aboutir à l'écriture de plusieurs algorithmes décrivant les bonnes stratégies (celles valables pour tous les nombres  $n$  et  $p$ ) et d'avoir un langage compréhensible par tous et par Xcas !

### 1.3 Une correction possible avec Xcas

Avec Xcas, on aura besoin de :

- la notion d'entrée et de sortie `saisir` et `afficher` ou de la notion de fonction,
- la notion d'affectation `:=`
- de l'instruction conditionnelle `si ...alors ...sinon ...fsi`
- la commande `iquorem(a, b)` qui renvoie le quotient et le reste de la division euclidienne de  $a$  par  $b$  :  
`iquorem(46, 7)` renvoie la liste `[6, 4]`

- la notion de séquence que l'on peut créer avec la commande `seq` ou l'opérateur infixé `$` :  
`seq(3, 5)` ou `3$5` renvoie `3, 3, 3, 3, 3`.

Il y a plusieurs stratégies possibles : on va en décrire 2 qui sont assez simples.

- Première stratégie,

On peut imaginer que  $n$  représente un nombre de cartes que l'on distribue à  $p$  joueurs jusqu'à épuisement des cartes.

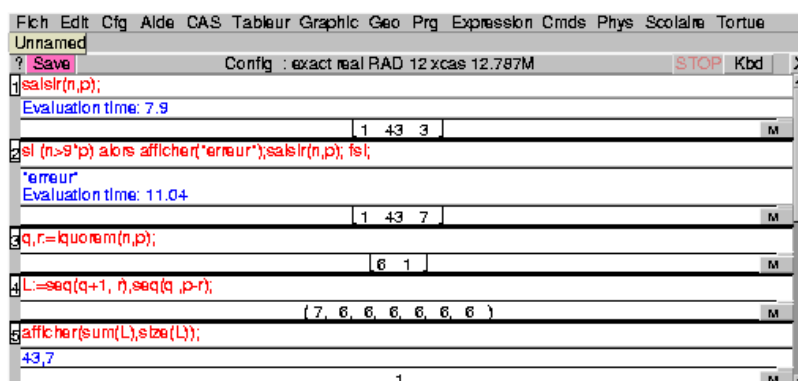
Si  $n = pq + r$  avec  $0 \leq r < p$  alors les  $r$  premiers joueurs auront  $q + 1$  cartes et les  $p - r$  derniers joueurs auront  $q$  cartes ( $n = (q + 1)r + q(p - r)$ ).

Avec Xcas, on tape :

```
saisir(n,p);
si (n>9*p) alors afficher("erreur");saisir(n,p); fsi;
q,r:=iquorem(n,p);
L:=seq(q+1, r),seq(q, p-r);
afficher(sum(L),size(L));
```

On entre par exemple 47 pour  $n$ , et 3 pour  $p$ , puis 47 pour  $n$ , et 7 pour  $p$ .

On obtient :



Ou encore, on écrit la fonction `Jeu1`. On prendra l'habitude de commencer le nom des fonctions par une Majuscule pour ne pas définir une fonction qui existe peut être déjà dans Xcas.

On tape :

```
Jeu1(n,p) := {
  local q,r,L;
  si (n>9*p) alors return "erreur" fsi;
  q,r:=iquorem(n,p);
  L:=(q+1) $ r,q $ (p-r);
  afficher(sum(L),size(L));
  return L;
};;
```

On renvoie la séquence  $L$  et on fait imprimer sa somme et sa taille pour vérifier qu'il n'y a pas d'erreurs.

On appuie sur `OK` et on obtient :

```

Prog Edit Ajouter      [next] [OK] [Save]
-----
Jeu1(n,p){
  local q,r;
  si (n>9*p) alors retu: "erreur fs ;
  q,r:=iquorem(
  L:=seq(q+1,r),seq(
  print(sum(L),si:
  retu: L
  };
// Parsing Jeu1
// Success compiling Jeu1
                                     Done
                                     M

```

On tape :

Jeu1(43,7)

On obtient :

7, 6, 6, 6, 6, 6, 6

On tape :

Jeu1(57,7)

On obtient :

9, 8, 8, 8, 8, 8, 8

On tape :

Jeu1(23,7)

On obtient :

4, 4, 3, 3, 3, 3, 3

– Deuxième stratégie,

On peut imaginer que  $n$  représente des billes que l'on doit mettre dans des boîtes. Chaque boîte doit contenir 9 billes. Si  $n = 9q + r$  avec  $0 \leq r < 9$ , on pourra, remplir les  $q$  premières boîtes, la boîte  $q + 1$  contiendra  $r$  billes et les  $p - q - 1$  dernières boîtes contiendront 0 billes.

On écrit la fonction `Jeu2` qui renvoie la séquence `L` et on fait imprimer la somme et la taille de `L` pour vérifier qu'il n'y a pas d'erreurs :

```

Jeu2(n,p) := {
  local q,r,L;
  q,r:=iquorem(n,9);
  si (n>9*p) alors return "erreur" fsi;
  L:=seq(9,q),r,seq(0,p-q-1);
  print(sum(L),size(L));
  return L;
};

```

On tape :

Jeu2(43,7)

On obtient :

9, 9, 9, 9, 7, 0, 0

On tape :

Jeu2(57,7)

On obtient :

9, 9, 9, 9, 9, 9, 3

On tape :

Jeu2(23,7)

On obtient :

9, 9, 5, 0, 0, 0, 0



## Chapitre 2

# Jeu sur la notion de boucle

### 2.1 Description

Le meneur de jeux choisit une séquence  $(a_0, a_1, \dots, a_9)$  de 10 nombres et il marque chaque nombre sur un carton. Il montre ces nombres les uns après les autres à la classe.

Chaque élève doit trouver une stratégie pour trouver la somme  $a_0 + a_1 + a_2 \dots + a_8 + a_9$  de cette séquence : il n'a pas le droit de noter ces nombres il a juste le droit de noter le résultat.

Chaque élève doit trouver une nouvelle stratégie pour dire si la somme  $a_0 + a_1 + a_2 \dots + a_8 + a_9$  est paire ou impaire.

Chaque élève doit trouver une stratégie pour trouver la somme alternée  $a_0 - a_1 + a_2 \dots + a_8 - a_9$  de cette séquence.

### 2.2 La stratégie

On doit faire la somme au fur et à mesure : on garde en tête  $a_0$ , puis  $a_0 + a_1 \dots$  Pour dire si la somme  $a_0 + a_1 + a_2 \dots + a_8 + a_9$  est paire ou impaire, il est inutile de calculer cette somme, il suffit de retenir à chaque étape la parité de la somme par exemple en représentant un nombre pair par 0 et un nombre impair par 1...

Pour avoir la somme alternée c'est un peu plus difficile car il faut savoir si on doit ajouter ou retrancher...

### 2.3 Une correction possible avec Xcas

```
Jeusomme(L) := {
  local S, j;
  S:=0;
  pour j de 0 jusque 9 faire
    S:=S+L[j];
  fpour;
  return S;
};
```

```

Jeupari(L) := {
  local S, j;
  S := 0;
  pour j de 0 jusque 9 faire
    S := S + L[j] mod 2 ;
  fpour;
  return S;
};

```

**On tape :**

```
Jeusomme(seq(2^n, (n=0..9)))
```

**On obtient :**

```
1023
```

**On tape :**

```
L := seq(2^n, (n=0..9))
```

**On obtient :**

```
1, 2, 4, 8, 16, 32, 64, 128, 256, 512
```

**On tape :**

```
Jeupari(seq(2^n, (n=0..9)))
```

**On obtient :**

```
1 % 2
```

**Donc la somme est impaire.**

**On tape :**

```
Jeusomme(seq((-2)^n, (n=0..9)))
```

**On obtient :**

```
-341
```

**On tape :**

```
LL := seq((-2)^n, (n=0..9))
```

**On obtient :**

```
1, -2, 4, -8, 16, -32, 64, -128, 256, -512
```

## Chapitre 3

# Jeu sur la notion de condition

### 3.1 Description

Le meneur de jeux choisit une séquence de 10 nombres  $(a_0, a_1, \dots, a_9)$  et il marque chaque nombre sur un carton. Il montre ces nombres les uns après les autres à la classe.

Chaque élève doit trouver une stratégie pour trouver le maximum de cette séquence.

Chaque élève doit trouver une stratégie pour trouver le minimum de cette séquence.

Chaque élève doit trouver une stratégie pour trouver le maximum et le minimum de cette séquence.

### 3.2 La stratégie

Pour trouver le maximum, on doit comparer le nouveau nombre avec le maximum des nombres précédents : on garde donc en tête un nombre  $M$  qui est le maximum provisoire et qui vaut au départ  $a_0$ , puis, qui sera maximum de  $a_0$  et  $a_1$  etc...

Pour trouver le minimum, on doit comparer le nouveau nombre avec le minimum des nombres précédents : on garde donc en tête un nombre  $m$  qui est le minimum provisoire et qui vaut au départ  $a_0$ , puis, qui sera minimum de  $a_0$  et  $a_1$  etc...

Pour trouver le maximum et le minimum, on doit garder en tête un nombre  $M$  et un nombre  $m$  comparer le nouveau nombre avec  $M$  et  $m$  et éventuellement mettre soit  $M$  soit  $m$  à jour.

### 3.3 Une correction possible avec Xcas

```
Jeumax(L) := {
  local M, j, n;
  M:=L[0];
  pour j de 1 jusque 9 faire
    n:=L[j];
    si (M<n) alors M:=n fsi;
  fpour;
  return M;
};;
```

**On tape :**

```
Jeumax(seq((-2)^n+3n, (n=0..9)))
```

**On obtient :**

```
280
```

**On tape :**

```
L3 :=seq((-2)^n+3n, (n=0..9))
```

**On obtient :**

```
1, 1, 10, 1, 28, -17, 82, -107, 280, -485
```

```
Jeumaxmin(L) := {
  local M, m, j, n;
  M := L[0];
  m := L[0];
  pour j de 1 jusque 9 faire
    n := L[j];
    si (M < n) alors
      M := n;
    sinon
      si (m > n) alors m := n; fsi;
    fsi;
  fpour;
  return M, m;
};;
```

**On tape :**

```
Jeumaxmin(seq((-2)^n+3n, (n=0..9)))
```

**On obtient :**

```
280, -485
```

**On tape :**

```
L3 :=seq((-2)^n+3n, (n=0..9))
```

**On obtient :**

```
1, 1, 10, 1, 28, -17, 82, -107, 280, -485
```

## Chapitre 4

# Vue d'ensemble de Xcas pour le programmeur

En vous référant au menu Aide->Manuels->Algorithmes vous aurez plus de détails sur l'algorithmique et Xcas, toutefois cette introduction est suffisante pour commencer.

Il y a aussi beaucoup d'exemples traités dans les autres manuels....

### 4.1 Installation de Xcas

Le programme Xcas est un logiciel libre écrit en C++, (disponible sous licence GPL). La version à jour se récupère sur :

`http://www-fourier.ujf-grenoble.fr/~parisse/giac_fr.html` ou

`ftp://fourier.ujf-grenoble.fr/xcas`

où l'on trouve le code source (`giac.tgz`) ou des versions précompilées pour Linux (PC ou ARM), Windows, Mac OS.

### 4.2 Les différents modes

Xcas propose un mode de compatibilité avec Maple, MuPAD et la TI89/92 : pour cela, il suffit de le spécifier dans `Prog style` du menu de configuration du cas (bouton `Config` ou menu `Cfg->Configuration` du CAS) ou avec le menu `Cfg->Mode (syntax)`. On peut choisir, en cliquant sur la flèche située à côté de `Prog style` : Xcas ou Maple ou MuPAD ou TI89/92.

On a aussi la possibilité d'importer une session Maple ou une archive TI89/92 en choisissant `Importer` du menu `Fich`, ou importer dans un niveau éditeur de programmes un fichier écrit en syntaxe Maple, Mupad ou TI89/92 par le menu `Prog->Insérer`.

On présente ici le mode Xcas qui est proche de la syntaxe C. On a aussi la possibilité d'avoir toutes les instructions en français de façon à être proche du langage Algorithmique.

### 4.3 Éditer, sauver, exécuter un programme avec la syntaxe `Xcas`

On écrit directement un script (i.e. une suite de commandes séparées par `;` ou par `:` ; ou plusieurs programmes ou fonctions séparées par `;` ou par `:` ; dans un niveau éditeur de programmes (que l'on ouvre avec `Alt+p`).

Depuis un niveau éditeur de programmes, on peut tester facilement si le programme est syntaxiquement correct grâce au bouton `OK` : la ligne où se trouve la faute de syntaxe est indiquée en bleu dans la zone intermédiaire.

On corrige les fautes lorsqu'il y en a...

Quand le programme est syntaxiquement correct, il y a dans la zone intermédiaire `Success compiling ...` et on a le programme en réponse ou simplement `Done` lorsqu'on a terminé l'écriture du programme par `:` ;. On peut alors exécuter le programme dans une ligne de commande.

Vous sauvez le programme avec le bouton `Save` du niveau éditeur de programmes sous le nom que vous voulez lui donner en le terminant par le suffixe `.cxx` (ce nom s'inscrit alors à côté du bouton `Save` du niveau éditeur de programmes. Si ensuite, vous voulez lui donner un autre nom il faut le faire avec le menu `Prog` sous-menu `Sauver` comme de l'éditeur de programmes.

### 4.4 Débugger un programme avec la syntaxe `Xcas`

Pour utiliser le débogueur, il faut que ce programme soit syntaxiquement correct : vous avez par exemple un programme syntaxiquement correct, mais qui ne fait pas ce qu'il devrait faire, il faut donc le corriger.

Avec le débogueur, on a la possibilité d'exécuter le programme au pas à pas (`sst`), ou d'aller directement (`cont`) à une ligne précise marquée par un point d'arrêt (`break`), de voir (`voir` ou `watch`) les variables que l'on désire surveiller, d'exécuter au pas à pas les instructions d'une fonction utilisateur utilisée dans le programme (dans ou `sst_in`), ou de sortir brutalement du débogueur (`tuer` ou `kill`).

On tape : `debug(nom _du_programme(valeur_des_arguments))`.

Il faut bien sûr que le programme soit validé :

- si le programme est dans un niveau éditeur de programme, on appuie sur `OK` pour le compiler, on corrige les fautes de syntaxe éventuelles et on appuie sur `OK` jusqu'à obtenir `Success compiling...`
- si le programme qui est syntaxiquement correct se trouve dans un fichier, on tape : `read("toto")` si `toto` est le nom du fichier où se trouve ce programme.

Par exemple, si `pgcd` a été validé, on tape :

```
debug(pgcd(15,25))
```

L'écran du débogueur s'ouvre : il est formé par trois écrans séparés par une ligne `eval` et une barre de boutons `sst`, `dans`, `cont`...

1. dans l'écran du haut, le programme source est écrit et la ligne en surbrillance sera exécutée grâce au bouton `sst`.

2. dans la ligne `eval`, `Xcas` marque automatiquement l'action en cours par exemple `sst`. Cette ligne permet aussi de faire des calculs dans l'environnement du programme ou de modifier une variable, par exemple on peut y écrire `a :=25` pour modifier la valeur de `a` en cours de programme,
3. dans l'écran du milieu, on trouve, le programme, les points d'arrêts, le numéro de la ligne du curseur.
4. une barre de boutons `sst`, `dans`, `cont` . . .
  - `sst` exécute la ligne courante (celle qui est en surbrillance) sans entrer dans les fonctions et met en surbrillance l'instruction suivante,
  - `dans` ou `sst_in` exécute la ligne courante (celle qui est en surbrillance) en entrant dans les fonctions utilisées dans le programme et qui ont été définies précédemment par l'utilisateur, puis met en surbrillance l'instruction suivante du programme en incluant les instructions de la fonction. Cela permet ainsi d'exécuter pas à pas les instructions de cette fonction.
  - `cont` exécute les instructions du programme situées entre la ligne courante et la ligne d'un point d'arrêt et met en surbrillance cette ligne,
  - `tuer` ou `kill` ferme brutalement l'écran du débogueur.
  - Attention** il faut fermer l'écran du débogueur pour pouvoir utiliser `Xcas`.
  - `break` ajoute un point d'arrêt. Les points d'arrêts permettent d'aller directement à un point précis avec le bouton `cont`. On marque les points d'arrêts grâce au bouton `break` ou à la commande `breakpoint` d'arguments le nom du programme et le numéro de la ligne où l'on veut un point d'arrêt : par exemple `breakpoint (pgcd, 3)`. Pour faciliter son utilisation, il suffit de cliquer dans l'écran du haut sur la ligne où l'on veut le point d'arrêt pour avoir : `breakpoint` dans la ligne `eval`, avec le nom du programme et le bon numéro de ligne, puis de valider la commande. Il suffit donc de cliquer et de valider !
  - `rmbrk` enlève un point d'arrêt. On doit, pour réutiliser d'autres points d'arrêts, d'effacer les points d'arrêts utilisés précédemment avec la commande `rmbreakpoint` qui a les mêmes arguments que `breakpoint`. Là encore, pour faciliter son utilisation, il suffit de cliquer sur la ligne où l'on veut enlever le point d'arrêt pour avoir : `rmbreakpoint` dans la ligne de commande, avec le nom du programme et le bon numéro de ligne.
  - Attention** si il n'y a pas de point d'arrêt à cet endroit `Xcas` en mettra un !
  - `voir` ou `watch` ajoute la variable que l'on veut voir évoluer. Si on ne se sert pas de `voir` ou `watch` toutes les variables locales et tous les arguments du programme sont montrées. Si on se sert de `voir` ou `watch` seules les variables désignées seront montrées : on appuie sur le bouton `voir` ou `watch` et la commande `watch` s'écrit dans la ligne d'évaluation `eval`. On tape alors, les arguments de `watch` qui sont les noms des variables que l'on veut surveiller, par exemple : `watch (b, r)` et on valide la commande.
  - `rmwch` efface les variables désignées précédemment avec `watch` et que l'on ne veut plus voir, par exemple : `rmwatch (r)`.
5. dans l'écran du bas, on voit soit l'évolution de toutes les variables locales et de tous les arguments du programme, soit l'évolution des variables désignées par `watch`.



## Chapitre 5

# Les instructions avec la syntaxe

Xcas

### 5.1 Les commentaires

Les commentaires sont des chaînes de caractères, ils sont précédés de `//` ou sont parenthésés par `/* */`

### 5.2 Le bloc

Une `action` ou `bloc` est une séquence d'une ou plusieurs instructions. Quand il y a plusieurs instructions il faut les parenthéser avec `{ }` et séparer les instructions par un point virgule `( ; )`. Un `bloc` est donc parenthésé par `{ }` et commence éventuellement par la déclaration des variables locales (`local . . .`).

### 5.3 Les variables globales et les variables locales

Voir aussi ?? Les variables sont les endroits où l'on peut stocker des valeurs, des nombres, des expressions, des objets.

Le nom des variables est formé par une suite de caractères et commence par une lettre : attention on n'a pas droit aux mots réservés ...ne pas utiliser par exemple la variable `i` dans un `for` ou un `pour` car `i` représente le nombre complexe de module 1 et d'argument  $\frac{\pi}{2}$ .

L'affectation se fait avec `:=` (par exemple `a:=2; b:=a;`).

### 5.4 Les programmes et les fonctions

- Les paramètres sont mis après le nom du programme ou de la fonction entre parenthèses : par exemple `f(a,b) := . . .`. Ces paramètres sont initialisés lors de l'appel du programme ou de la fonction et se comportent comme des variables locales.
- L'affectation se fait avec `:=` (par exemple `a:=2; b:=a;`),
- Les entrées se font par passage de paramètres ou avec `input`,

- Les sorties se font avec `print`,
- Il n’y a pas de distinction entre programme et fonction : la valeur d’une fonction est précédée du mot réservé `return`.

**Remarque** `return` n’est pas obligatoire car `Xcas` renvoie toujours la valeur de la dernière instruction, mais `return` est très utile car il fait sortir de la fonction : les instructions situées après `return` ne sont jamais effectuées.

## 5.5 Les tests

Avec le langage `Xcas`, les tests ont soit une syntaxe similaire au langage `C++` soit une version française proche du langage algorithmique.

Pour les tests, les syntaxes admises sont :

- `if (condition) instruction;`  
on met `{..}` lorsqu’il faut faire plusieurs instructions :  
`if (condition) {instructions}`  
ou  
si `condition` alors `instructions` fsi  
on teste la condition : si elle est vraie, on fait les instructions et si elle est fausse on ne fait rien c’est à dire on passe aux instructions qui suivent le `if` ou le `si`.

Par exemple :

```
testif1(a,b) := {
  if (a<b)
    b:=b-a;
  return [a,b];
};
```

ou

```
testsil(a,b) := {
  si a<b alors
    b:=b-a;
  fsi;
  return [a,b];
};
```

et on a :

```
testif1(3,13)=testsil(3,13)=[3,10]
testif1(13,3)=testsil(13,3)=[13,3]
```

- `if (condition) instruction1; else instruction2;`  
on met `{..}` lorsqu’il faut faire plusieurs instructions :  
`if (condition) {instructions1} else {instructions2}`  
ou  
si `condition` alors `instructions1` sinon `instructions2`  
fsi  
on teste la condition : si elle est vraie, on fait les `instructions1` et si elle est fausse on fait les `instructions2`.

Par exemple :

```
testif(a,b) := {
  if (a==10 or a<b)
```

```

        b:=b-a;
else
        a:=a-b;
return [a,b];
};
ou
testsi(a,b):={
si a==10 or a<b alors
        b:=b-a;
sinon
        a:=a-b;
fsi;
return [a,b];
};
et on a :
testif(3,13)=testsi(3,13)=[3,10]
testif(13,3)=testsi(13,3)=[10,3]
testif(10,3)=testsi(10,3)=[10,-7]

```

## 5.6 Les boucles

Avec le langage Xcas, les boucles ont soit une syntaxe similaire au langage C++ soit une version française proche du langage algorithmique.

Pour les boucles, les syntaxes admises sont :

- la boucle `for` ou `pour` qui permet de faire des instructions un nombre de fois qui est connu :

```
for (init;condition;increment) instruction;
```

on met {..} lorsqu'il faut faire plusieurs instructions :

```
for (init;condition;increment) {instructions}
```

Le plus souvent (*init;condition;increment*) s'écrit en utilisant une variable (par exemple *j* ou *k*...mais pas *i* qui désigne un nombre complexe!!!). Cette variable sera initialisée dans *init*, utilisée dans *condition* et incrémentée dans *increment*, on écrit par exemple :

(*j:=1;j<=10;j++*) (l'incrément ou le pas est de 1) ou

(*j:=10;j>=1;j--*) (l'incrément ou le pas est de -1) ou

(*j:=1;j<=10;j:=j+2*) (l'incrément ou le pas est de 2)

ou

pour *j* de 1 jusque 10 faire *instructions* fpour ;

ou

pour *j* de 10 jusque 1 pas -1 faire *instructions* fpour ;

ou

pour *j* de 1 jusque 10 pas 2 faire *instructions* fpour ;

On initialise *j* puis on teste la condition :

- si elle est vraie, on fait les instructions puis on incrémente *j*, puis, on teste la condition : si elle est vraie, on fait les instructions etc...
- si elle est fausse on ne fait rien c'est à dire on passe aux instructions qui suivent le `for` ou le `pour`.

Par exemple :

```
testfor1(a,b) :={
local j,s:=0;
for (j:=a;j<=b;j++)
  s:=s+1/j^2;
return s;
};
```

ou

```
testpour1(a,b) :={
local j,s:=0;
pour j de a jusque b faire
  s:=s+1/j^2;
fpour;
return s;
};
```

Si  $a > b$ , l'instruction ou le bloc d'instructions du `for` ou du `pour` ne se fera pas et la fonction retournera 0,

Si  $a \leq b$  la variable  $j$  va prendre successivement les valeurs  $a, a+1, \dots, b$  (on dit que le pas est de 1) et pour chacune de ces valeurs l'instruction ou le bloc d'instructions qui suit sera exécuté. Par exemple `testfor1(1,2)` renverra  $1+1/4=5/4$ .

```
testfor2(a,b) :={
local j,s:=0;
for (j:=b;j>=a;j--)
  s:=s+1/j^2;
return s;
};
```

ou

```
testpour2(a,b) :={
local j,s:=0;
pour j de b jusque a pas -1 faire
  s:=s+1/j^2;
fpour;
return s;
};
```

Dans ce cas, si  $a \leq b$  la variable  $j$  va prendre successivement les valeurs  $b, b-1, \dots, a$  (on dit que le pas est de -1) et pour chacune de ces valeurs l'instruction ou le bloc d'instructions qui suit sera exécuté. Par exemple `testfor2(1,2)` renverra  $1/4+1=5/4$ .

```
testfor3(a,b) :={
local j,s:=0;
for (j:=a;j<=b;j:=j+3)
  s:=s+1/j^2;
return s;
};
```

ou

```
testpour3(a,b) :={
```

```

local j,s:=0;
pour j de a jusque b pas 3 faire
  s:=s+1/j^2;
fpour;
return s;
};

```

Dans ce cas, si  $a \leq b$  la variable  $j$  va prendre successivement les valeurs  $a, a+3, \dots, a+3k$  avec  $k$  le quotient de  $b-a$  par 3 ( $3k \leq b-a < 3(k+1)$ ) (on dit que le pas est de 3) et pour chacune de ces valeurs l'instruction ou le bloc d'instructions qui suit sera exécuté. Par exemple `testfor3(1,5)` renverra  $1+1/16=17/16$ .

- la boucle `while` ou `tantque` permet de faire plusieurs fois des instructions avec une condition d'arrêt au début de la boucle : `while (condition) instruction;`  
on met `{..}` lorsqu'il faut faire plusieurs instructions :  
`while (condition) {instructions}`  
ou  
`tantque condition faire instructions ftantque;`  
On teste la condition :
  - si elle est vraie, on fait les instructions puis, on teste la condition : si elle est vraie, on fait les instructions etc...
  - si elle est fausse on ne fait rien c'est à dire on passe aux instructions qui suivent le `while` ou le `tantque`.

Par exemple :

```

testwhile(a,b) :={
while (a==10 or a<b)
  b:=b-a;
return [a,b];
};
ou
testtantque(a,b) :={
tantque a==10 or a<b faire
  b:=b-a;
ftantque
return [a,b];
};

```

**Un exemple : le PGCD d'entiers**

```

pgcd(a,b) :={
local r;
while (b!=0) {
  r:=irem(a,b);
  a:=b;
  b:=r;
}
return a;
};

```

```

ou
pgcd(a,b) :={
  local r;
  tantque b!=0 faire
    r:=irem(a,b);
    a:=b;
    b:=r;
  ftantque;
  return a;
};

```

- La boucle `repeat` ou `repetet` permet de faire plusieurs fois des instructions avec une condition d'arrêt à la fin de la boucle :

```

repeat instructions until condition;
ou
repetet instructions jusqu_a condition;
ou
repetet instructions jusqu_a condition;

```

On fait les instructions, puis on teste la condition :

- si elle est vraie, on fait les instructions puis, on teste la condition etc...
- si elle est fausse on passe aux instructions qui suivent le `repeat` ou le `repetet`.

Par exemple :

```

f() :={
  local a;
  repeat saisir("entrez un reel entre 1 et 10",a); until a>=1 et
  return a;
}
ou
f() :={
  local a;
  repetet saisir("entrez un reel entre 1 et 10",a); jusqu_a a>=1 e
  return a;
}

```

## Chapitre 6

# Des exemples de programmes

### 6.1 Maximum de 3 nombres

On donne 3 nombres  $a, b, c$  trouver le maximum de ces 3 nombres.

On va décrire plusieurs méthodes :

- On peut utiliser la formule :  $\max(a, b, c) = \max(\max(a, b), c)$  et traduire cette formule par un algorithme.

On tape :

```
Max2(a, b) := {
  si a > b alors return a;
  sinon return b; fsi
};
Max3(a, b, c) := Max2(Max2(a, b), c)
```

- On gère tous les cas avec des si ...alors ...sinon...fsi.

On tape :

```
Maxi(a, b, c) := {
  local M;
  si a > b alors
    si a > c alors
      M := a;
    sinon M := c; fsi
  sinon
    si c > b alors
      M := c;
    sinon M := b; fsi
  fsi
  return M;
};
```

- On utilise return qui fait sortir de la boucle.

Ou bien on teste si  $a$  est plus grand que  $b$  et est plus grand que  $c$  et dans ce cas on renvoie  $a$ . Puis on teste la même chose avec  $b$  puis avec  $c$ .

On tape :

```
Max(a, b, c) := {
  si a >= b et a >= c alors return a; fsi
  si b >= a et b >= c alors return b; fsi
  si c >= a et c >= b alors return c; fsi
};
```

```
};
```

- On peut aussi écrire une procédure récursive pour traduire que on teste si  $a$  est plus grand que  $b$  et est plus grand que  $c$  et dans ce cas on renvoie  $a$  et sinon on fait la même chose avec  $b$  puis avec  $c$ .

On tape :

```
Maxr(a,b,c) := {
  si a >= b et a >= c alors return a; fsi
  Maxr(b,c,a);
};
```

Ici le programme est plus difficile à comprendre car : dans le cas où  $a < b$  ou  $a < c$ ,  $\text{Maxr}(a, b, c)$  appelle  $\text{Maxr}(b, c, a)$  qui dans le cas où  $b < a$  ou  $b < c$ , appelle  $\text{Maxr}(c, a, b)$

## 6.2 Ordonner 3 nombres

On peut modifier les programmes ci-dessus pour que le résultat soit les 3 nombres ordonnés par ordre décroissant.

- On trie d'abord  $a, b$ , puis on insère  $c$  dans la liste ordonnée  $a, b$ .

On tape :

```
Tri2(a,b) := {
  si a > b alors return a,b;
  sinon return b,a; fsi
};
Tri3(a,b,c) := {
  a,b := Tri2(a,b);
  si c >= a alors return c,a,b;
  si c < b alors return a,b,c;
  return a,c,b;
};
```

- On teste les 6 cas.

On tape :

```
Tri(a,b,c) := {
  si a >= b et b >= c alors return a,b,c; fsi
  si a >= c et c >= b alors return a,c,b; fsi
  si b >= c et c >= a alors return b,c,a; fsi
  si b >= a et b >= c alors return b,a,c; fsi
  si c >= a et a >= b alors return c,a,b; fsi
  si c >= b et b >= a alors return c,b,a; fsi
};
```

- On gère tous les avec des `si ...alors ...sinon...fsi`, mais en ordonnant au début  $a$  et  $b$ .

On tape :

```
Trii(a,b,c) := {
  local L,d;
  si a < b alors
  d:=a;a:=b;b:=d;
  fsi
```

```

si b>=c alors
  L:=a,b,c;
sinon
  si c>=a alors
    L:=c,a,b;
  sinon
    L:=a,c,b;
  fsi
fsi
return L;
};;
- On utilise la récursivité.
On tape :
Trirr(a,b,c) :={
si a>=b alors
  si b>=c alors return a,b,c;
  sinon
    si a>=c alors return a,c,b; fsi
  fsi
Trirr(b,c,a);
};;
Ou on tape :
Trir(a,b,c) :={
si a>=b et b>=c alors
  return a,b,c;
fsi
si a>=c et c>=b alors
  return a,c,b;
fsi
Trir(b,c,a);
};;

```

### 6.3 Équation d'une droite définie par 2 points

On clique 2 points  $A$  et  $B$  dans l'écran de géométrie en mode point. On cherche l'équation de la droite  $AB$ .

Si  $a_1, a_2$  sont les coordonnées de  $A$ , si  $b_1, b_2$  sont les coordonnées de  $B$ , on cherche  $a, b, c$  pour que la droite d'équation  $a \cdot x + b \cdot y + c = 0$  passe par  $A$  et  $B$ . Il faut donc résoudre en  $a, b, c$  :

$$aa_1 + ba_2 + c = 0$$

$$ab_1 + bb_2 + c = 0$$

Par soustraction membre à membre on a

$$a * (a_1 - b_1) + b * (a_2 - b_2) = 0$$

On choisit :  $a = a_2 - b_2$  et  $b = -a_1 + b_1$  et alors  $c = -a * a_1 - b * a_2$

On tape :

```

a1,a2 :=coordonnees (A) ;
b1,b2 :=coordonnees (B) ;
a :=a2-b2 ;
b :=-a1+b1 ;
c :=-a*a1-b*a2 print (a*x+b*y+c)
si (b!=0) alors y=-a/b*x-c/b ; sinon a*x+b*y+c=0 ; fsi
On peut aussi l'écrire sous la forme de la fonction Equation_droite :

```

```

Equation_droite (A, B) :={
  local a, a1, a2, b; b1, b2, c;
  a1, a2:=coordonnees (A) ;
  b1, b2:=coordonnees (B) ;
  a:=a2-b2;
  b:=-a1+b1;
  c:=-a*a1-b*a2
  si (b!=0) alors
    return y=-a/b*x-c/b;
  sinon
    return a*x+b*y+c=0;
  fsi
};;

```

**Remarque** Dans Xcas la commande `equation` existe et renvoie l'équation de l'objet géométrique donné en argument.

## 6.4 Un puzzle avec les pentaminos

### 6.4.1 L'activité

Les pentaminos s'obtiennent en juxtaposant 5 carrés unités : chaque carré doit avoir au moins un côté en commun avec un autre carré. Il y a 12 pentaminos car on s'autorise à retourner les pièces que l'on suppose colorées sur les 2 faces.

L'activité consiste :

- Dans un premier temps, à déterminer la forme des 12 pentaminos en insistant sur une recherche méthodique pour éviter les oublis. Les élèves pourront découper leurs dessins pour pouvoir par superposition éliminer les figures isométriques.
- Dans un deuxième temps, à programmer ces pièces. Pour pouvoir poser les pentaminos à un endroit précis du plan, il faut prévoir 3 paramètres : 2 paramètres  $a_1, a_2$  qui seront les coordonnées du point d'ancrage et qui permettront de changer la position de la pièce et un paramètre qui sera la couleur de la pièce. Il faut savoir que la couleur est codée par un entier  $n$  mais que l'on peut aussi utiliser le nom des couleurs élémentaires : 0 ou noir, 1 ou rouge, 2 ou vert, 3 ou jaune, 4 ou bleu, 5 ou magenta, 6 ou cyan, 7 ou blanc.

Ce travail permet de travailler sur le repérage d'un point dans le plan.

- Dans un troisième temps, de jouer avec ses pièces : par exemple de juxtaposer les 12 pentaminos pour obtenir un rectangle. On utilisera ensuite les

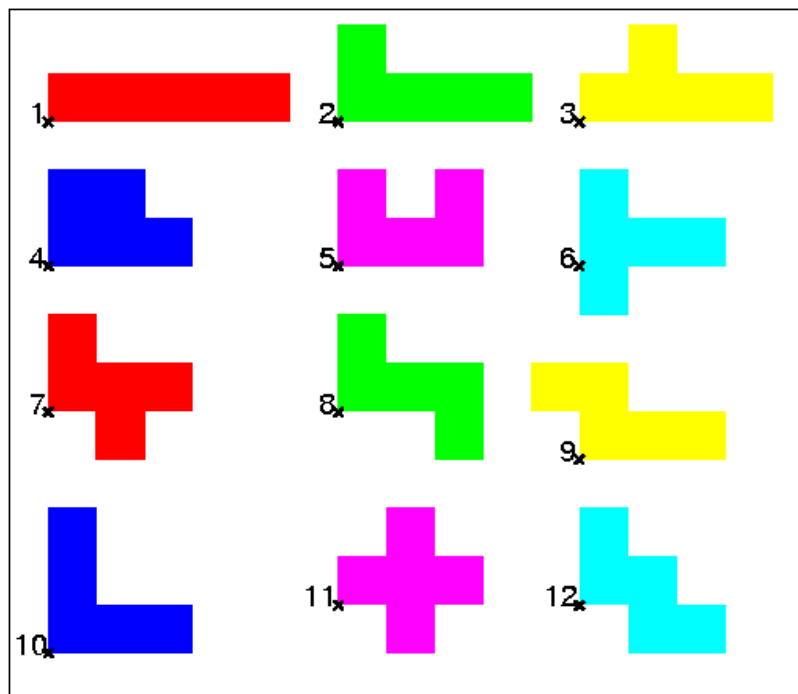
transformations `rotation` pour changer l'orientation et `symetrie` pour faire un retournement.

### Remarques

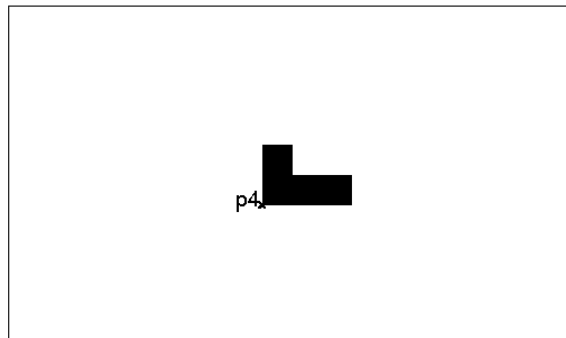
On peut éviter, au début, d'utiliser `rotation` et `symetrie` à condition de faire le puzzle et de programmer les pièces selon la position qu'elles ont dans le puzzle...  
On peut éviter, au début, d'utiliser des variables locales et des sous-procédures.

### 6.4.2 Programmes des pièces

Sur le dessin ci-après on a noté le point d'ancrage qui correspond aux paramètres `a1`, `a2` et le numéro de la pièce :



Pour obtenir les 12 pentaminos, on utilise l'instruction `carre` et `rectangle` et on définit la forme `p4` constituée de 4 carrés et que l'on retrouve dans plusieurs pentaminos :



On peut bien sûr n'utiliser que `carre` pour programmer les 12 pentaminos.  
On tape :

```
penta1(a1,a2,c) := {
  local P;
  P:=rectangle(point(a1,a2),point(a1+5,a2),1/5);
  return affichage(P,rempli+c);
}
;;
penta2(a1,a2,c) := {
  local P;
  P:=rectangle(point(a1,a2),point(a1+4,a2),1/4),
  carre(point(a1,a2+1),point(a1+1,a2+1));
  return affichage(P,rempli+c);
}
;;
penta3(a1,a2,c) := {
  local P;
  P:=rectangle(point(a1,a2),point(a1+4,a2),1/4),
  carre(point(a1+1,a2+1),point(a1+2,a2+1));
  return affichage(P,rempli+c);
}
;;
p4(a1,a2) := {
  local P;
  P:=rectangle(point(a1,a2),point(a1+3,a2),1/3),
  carre(point(a1,a2+1),point(a1+1,a2+1));
  return P;
};;
penta4(a1,a2,c) := {
  local P;
  P:=p4(a1,a2),carre(point(a1+1,a2+1),point(a1+2,a2+1));
  return affichage(P,rempli+c);
};;
penta5(a1,a2,c) := {
  local P;
```

```

P:=p4(a1,a2),carre(point(a1+2,a2+1),point(a1+3,a2+1));
return affichage(P,rempli+c);
};;
penta6(a1,a2,c):={
local P;
P:=p4(a1,a2),carre(point(a1,a2-1),point(a1+1,a2-1));
return affichage(P,rempli+c);
};;
penta7(a1,a2,c):={
local P;
P:=p4(a1,a2),carre(point(a1+1,a2-1),point(a1+2,a2-1));
return affichage(P,rempli+c);
};;
penta8(a1,a2,c):={
local P;
P:=p4(a1,a2),carre(point(a1+2,a2-1),point(a1+3,a2-1));
return affichage(P,rempli+c);
};;
penta9(a1,a2,c):={
local P;
P:=p4(a1,a2),carre(point(a1-1,a2+1),point(a1,a2+1));
return affichage(P,rempli+c);
};;
penta10(a1,a2,c):={
local P;
P:=p4(a1,a2),carre(point(a1,a2+2),point(a1+1,a2+2));
return affichage(P,rempli+c);
};;
penta11(a1,a2,c):={
local P;
P:=rectangle(point(a1,a2),point(a1+3,a2),1/3),
carre(point(a1+1,a2+1),point(a1+2,a2+1)),
carre(point(a1+1,a2-1),point(a1+2,a2-1));
return affichage(P,rempli+c);
};;
penta12(a1,a2,c):={
local P;
P:=carre(point(a1,a2+1),point(a1+1,a2+1)),
rectangle(point(a1,a2),point(a1+2,a2),1/2),
rectangle(point(a1+1,a2-1),point(a1+3,a2-1),1/2);
return affichage(P,rempli+c);
};;

```

On peut aussi ne pas introduire la variable locale P, mais quand même utiliser la sous-procédure p4 :

```

penta1(a1,a2,c):=
affichage(rectangle(point(a1,a2),point(a1+5,a2),1/5),
rempli+c);;

```

```

penta2(a1, a2, c) :=
    affichage(rectangle(point(a1, a2), point(a1+4, a2), 1/4),
               carre(point(a1, a2+1), point(a1+1, a2+1)),
               rempli+c) ;;
penta3(a1, a2, c) :=
    affichage(rectangle(point(a1, a2), point(a1+4, a2), 1/4),
               carre(point(a1+1, a2+1), point(a1+2, a2+1)),
               rempli+c) ;;
p4(a1, a2) := {
    local P;
    P := rectangle(point(a1, a2), point(a1+3, a2), 1/3),
           carre(point(a1, a2+1), point(a1+1, a2+1));
    return P;
} ;;
penta4(a1, a2, c) :=
    affichage(p4(a1, a2),
               carre(point(a1+1, a2+1), point(a1+2, a2+1)),
               rempli+c) ;;
penta5(a1, a2, c) :=
    affichage(p4(a1, a2),
               carre(point(a1+2, a2+1), point(a1+3, a2+1)),
               rempli+c) ;;
penta6(a1, a2, c) :=
    affichage(p4(a1, a2),
               carre(point(a1, a2-1), point(a1+1, a2-1)),
               rempli+c) ;;
penta7(a1, a2, c) :=
    affichage(p4(a1, a2),
               carre(point(a1+1, a2-1), point(a1+2, a2-1)),
               rempli+c) ;;
penta8(a1, a2, c) :=
    affichage(p4(a1, a2),
               carre(point(a1+2, a2-1), point(a1+3, a2-1)),
               rempli+c) ;;
penta9(a1, a2, c) :=
    affichage(p4(a1, a2),
               carre(point(a1-1, a2+1), point(a1, a2+1)),
               rempli+c) ;;
penta10(a1, a2, c) :=
    affichage(p4(a1, a2),
               carre(point(a1, a2+2), point(a1+1, a2+2)),
               rempli+c) ;;
penta11(a1, a2, c) :=
    affichage(rectangle(point(a1, a2), point(a1+3, a2), 1/3),
               carre(point(a1+1, a2+1), point(a1+2, a2+1)),
               carre(point(a1+1, a2-1), point(a1+2, a2-1)),
               rempli+c) ;;
penta12(a1, a2, c) :=

```

```

affichage(carre(point(a1,a2+1),point(a1+1,a2+1)),
           rectangle(point(a1,a2),point(a1+2,a2),1/2),
           rectangle(point(a1+1,a2-1),point(a1+3,a2-1),1/2),
           rempli+c);

```

Voici, à titre d'exemple pour le professeur, ce qui a été tapé pour obtenir les pièces avec leurs légendes : `depart()` permet de dire où se font les points d'ancrage des pièces et `pieces()` dessine les pièces avec leur point d'ancrage.

On peut aussi ne pas définir de fonctions et mettre seulement la suite des instructions par exemple :

```

point(-8,5,affichage=quadrant2+epaisseur_point_2,legend="1"),
penta1(-8,5,1)

```

définit le pentamino1 avec son point d'ancrage.

```

depart() := {
point(-8,5,affichage=quadrant2+epaisseur_point_2,legend="1"),
point(-2,5,affichage=quadrant2+epaisseur_point_2,legend="2"),
point(3,5,affichage=quadrant2+epaisseur_point_2,legend="3"),
point(-8,2,affichage=quadrant2+epaisseur_point_2,legend="4"),
point(-2,2,affichage=quadrant2+epaisseur_point_2,legend="5"),
point(3,2,affichage=quadrant2+epaisseur_point_2,legend="6"),
point(-8,-1,affichage=quadrant2+epaisseur_point_2,legend="7"),
point(-2,-1,affichage=quadrant2+epaisseur_point_2,legend="8"),
point(3,-2,affichage=quadrant2+epaisseur_point_2,legend="9"),
point(-8,-6,affichage=quadrant2+epaisseur_point_2,legend="10"),
point(-2,-5,affichage=quadrant2+epaisseur_point_2,legend="11"),
point(3,-5,affichage=quadrant2+epaisseur_point_2,legend="12");
};
pieces() := {
penta1(-8,5,rouge),penta2(-2,5,vert),penta3(3,5,jaune),
penta4(-8,2,bleu),penta5(-2,2,magenta),penta6(3,2,cyan),
penta7(-8,-1,rouge),penta8(-2,-1,vert),penta9(3,-2,jaune),
penta10(-8,-6,bleu),penta11(-2,-5,magenta),penta12(3,-5,cyan),
depart();
};

```

### 6.4.3 Le puzzle du rectangle 6×10

La construction du rectangle 6×10 en juxtaposant les 12 pentaminos est le "Problème Fondamental du Jeu de Pentaminos". Il y a 2339 solutions, mais pour obtenir une solution, ce n'est pas si simple !

Voici deux solutions et leurs programmes au "Problème Fondamental". On remarquera qu'avec Xcas les objets géométriques perdent leurs attributs lorsqu'ils sont transformés par les instructions `rotation` ou `symetrie`, on doit donc les spécifier avec une commande `affichage`, par exemple on mettra :

```

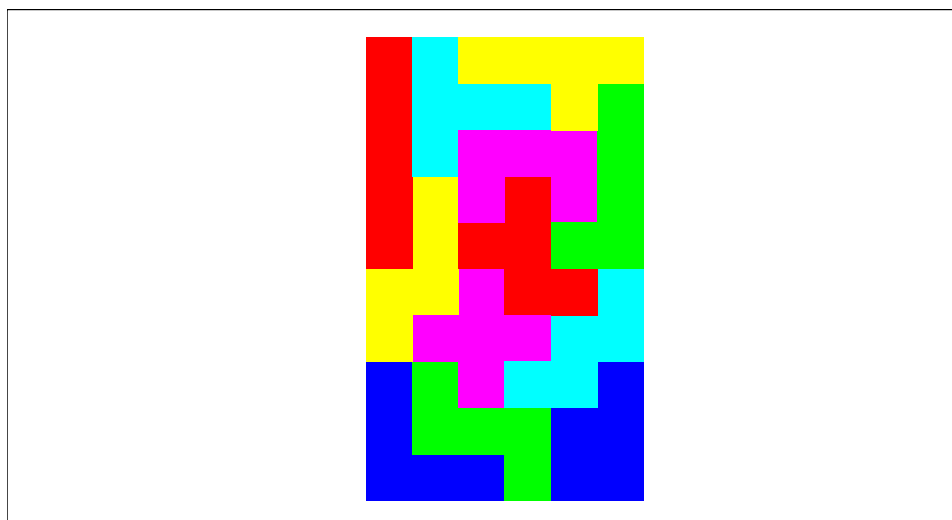
affichage(rotation(6,pi/2,penta4(6,0,4)),rempli+4) ou
affichage(rotation(6,pi/2,penta4(6,0,bleu)),rempli+bleu)

```

On tape les fonctions `puzzle1()` et `puzzle2()` mais on peut aussi ne pas

définir de fonctions et mettre seulement la suite des instructions ce qui permet de travailler avec des élèves qui n'ont pas vu les fonctions.

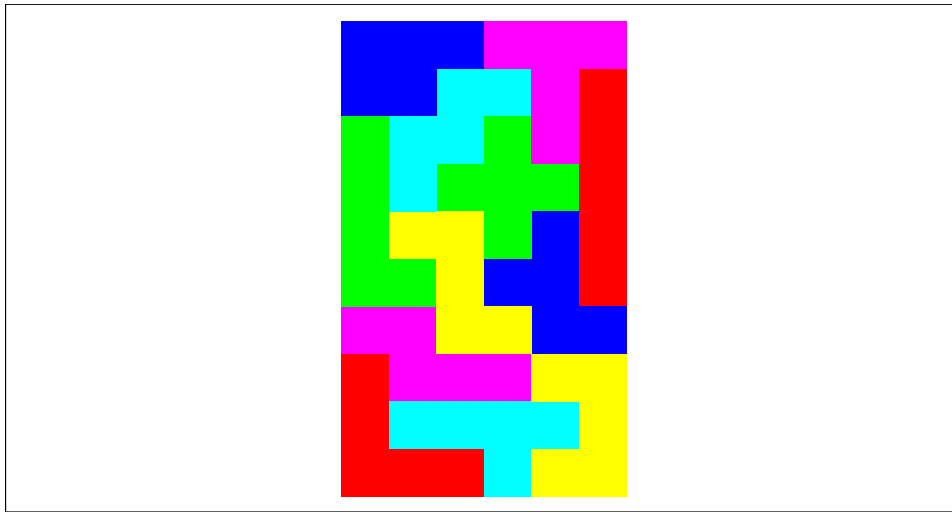
Voici le puzzle1 :



On tape :

```
puzzle1() := {
  penta10(0,0,bleu), penta8(1,1,vert),
  affichage(rotation(6,pi/2,penta4(6,0,4)), rempli+bleu),
  penta11(1,3,magenta),
  affichage(rotation(2+4*i,pi/2,penta9(2,4,3)), rempli+jaune),
  affichage(rotation(5+2*i,pi/2,penta12(5,2,6)), rempli+cyan),
  affichage(rotation(1+5*i,pi/2,penta1(1,5,1)), rempli+rouge),
  penta6(1,8,cyan),
  affichage(symetrie(droite(3,3+i),
    rotation(3+4*i,pi/2,penta7(3,4,1))), rempli+rouge),
  affichage(symetrie(droite(8*i,3+8*i),penta5(2,8,5)),
    rempli+magenta),
  affichage(rotation(6+5*i,pi/2,penta2(6,5,2)), rempli+vert),
  affichage(rotation(6+10*i,pi,penta3(6,10,3)), rempli+jaune);
};;
```

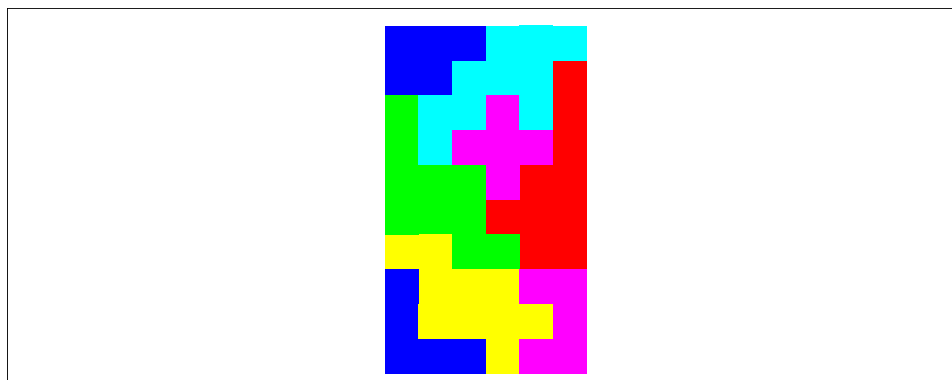
Voici le puzzle2 :



On tape :

```
puzzle2 () :={
penta10 (0,0, rouge), penta9 (1,2, magenta),
affichage (rotation (5+2*i, pi, penta3 (5,2,6)), rempli+cyan),
affichage (rotation (6, pi/2, penta5 (6,0,3)), rempli+jaune),
affichage (symetrie (droite (0,i),
rotation (4*i, pi/2, penta2 (0,4,2))), rempli+vert),
affichage (symetrie (droite (4,4+i),
rotation (4+3*i, pi/2, penta7 (4,3,4))), rempli+bleu),
affichage (symetrie (droite (2,2+i),
rotation (2+3*i, pi/2, penta8 (2,3,3))), rempli+jaune),
penta11 (2,6, vert),
affichage (rotation (6+4*i, pi/2, penta1 (6,4,1)), rempli+rouge),
affichage (rotation (2+9*i, -pi/2, penta12 (2,9,6)), rempli+cyan),
affichage (symetrie (droite (10*i, 1+10*i), penta4 (0,10,4)),
rempli+bleu),
affichage (rotation (4+10*i, -pi/2, penta6 (4,10,5)),
rempli+magenta);
};
```

ou bien si veut que pour  $n=1 \dots 6$ , la pièce  $n$  et la pièce  $n+6$  soient de couleur  $n$ , le puzzle2 devient le puzzle2b :



```

puzzle2b() :={
penta10(0,0,4),penta9(1,2,3),
affichage(rotation(5+2*i,pi,penta3(5,2,3)),rempli+3),
affichage(rotation(6,pi/2,penta5(6,0,5)),rempli+5),
affichage(symetrie(droite(0,i),
rotation(4*i,pi/2,penta2(0,4,2))),rempli+2),
affichage(symetrie(droite(4,4+i),
rotation(4+3*i,pi/2,penta7(4,3,1))),
rempli+1),
affichage(symetrie(droite(2,2+i),
rotation(2+3*i,pi/2,penta8(2,3,2))),
rempli+2),
penta11(2,6,5),
affichage(rotation(6+4*i,pi/2,penta1(6,4,1)),rempli+1),
affichage(rotation(2+9*i,-pi/2,penta12(2,9,6)),rempli+6),
affichage(symetrie(droite(10*i,1+10*i),penta4(0,10,4)),
rempli+4),
affichage(rotation(4+10*i,-pi/2,penta6(4,10,6)),rempli+6);
};

```

### Remarque

Tous les rectangles de dimension  $3 \times 20$ ,  $4 \times 15$ ,  $5 \times 12$  et  $6 \times 10$  peuvent être construits :

- de 2 façons pour le rectangle  $3 \times 20$ ,
- de 368 façons pour le rectangle  $4 \times 15$ ,
- de 1010 façons pour le rectangle  $5 \times 12$ ,
- de 2339 façons pour le rectangle  $6 \times 10$ .

A vous de jouer!!!!

## 6.5 Trouver le $n$ ième terme d'une suite récurrente

### 6.5.1 Un exemple

Soit  $f$  une fonction, on définit la suite  $u_n$  par  $u_0$  et par  $u_n = f(u_{n-1})$  pour  $n > 0$ . Par exemple  $f(x) = -7x/10 + 2$  et  $u_0 = 3$ .

On veut calculer les premiers termes de cette suite.

On tape :

$$f(x) := -7x/10 + 2$$

puis on tape :

3

puis on tape

$$f(\text{ans}())$$

et on obtient  $41/10$ , puis on tape

$$f(\text{ans}())$$

et on obtient  $487/100$ , etc....

### 6.5.2 Interêts d'un placement

On place 10 000 euros à un taux annuel de 4%. Quelle est la somme que l'on obtient au bout de 5 ans ? au bout de 10 ans ?

Une somme  $S$  devient  $S * (1 + 4/100)$  au bout de 1 an,

Une somme  $S$  devient  $S * (1 + 4/100)^2$  au bout de 2 ans,

...

Une somme  $S$  devient  $S * (1 + 4/100)^5$  au bout de 5 ans,

Une somme  $S$  devient  $S * (1 + 4/100)^{10}$  au bout de 10 ans,

On tape : `evalf(10000*(1+4/100)^5)`

On obtient 12166.529024

On tape : `evalf(10000*(1+4/100)^10)`

On obtient 14802.4428492

### 6.5.3 Mensualités d'un emprunt

On veut emprunter  $S = 10000$  euros sur 5 ans (respectivement 60 mois) à un taux annuel de 4% (respectivement à un mensuel de 0.33 %). Combien doit-on rembourser annuellement (mensuellement) ?

Sur 5 ans, on fait 5 remboursements annuels égaux à  $r$ .

Au bout de 1 an, la somme  $S$  devient :

$$S * (1 + 4/100) - r,$$

Au bout de 2 ans, la somme  $S$  devient :

$$(S * (1 + 4/100) - r) * (1 + 4/100) - r = S * (1 + 4/100)^2 - r * (1 + 4/100) - r,$$

Au bout de 3 ans, la somme  $S$  devient :

$$(S * (1 + 4/100)^2 - r * (1 + 4/100) - r) * (1 + 4/100) - r =$$

$$S * (1 + 4/100)^3 - r * (1 + 4/100)^2 - r * (1 + 4/100) - r,$$

...

Au bout de 5 ans, la somme  $S$  devient :

$$S * (1 + 4/100)^5 - r * ((1 + 4/100)^4 + (1 + 4/100)^3 + (1 + 4/100)^2 + (1 + 4/100) + 1)$$

i.e.

$$S * (1 + 4/100)^5 - r * ((1 + 4/100)^5 - 1) / (4/100),$$

Si on veut avoir tout remboursé au bout de 5 ans, il faut résoudre :

$$S * (1 + 4/100)^5 - r * ((1 + 4/100)^5 - 1) / 4 * 100 = 0.$$

On tape :

```
evalf(solve(10000*(1+4/100)^5-r*((1+4/100)^5-1)/4*100,r))
```

On obtient [2246.27113493]

On aura remboursé en tout :  $2246.27 \times 5 = 11231.35$  euros.

On vous propose de rembourser mensuellement 187 euros par mois.

Est-ce correct ?

On cherche le taux mensuel  $t$  de cet emprunt.

Au bout de 1 mois, la somme  $S$  devient  $S * (1 + t) - r$ ,

Au bout de 2 mois, la somme  $S$  devient  $(S * (1 + t)^2 - r * (1 + t) - r$ ,

.....

Au bout de 60 mois, la somme  $S$  devient :

$(S(1+t)^{60} - r * ((1+t)^{59} \dots (1+t) + 1)$  soit  $S(1+t)^{60} - r * ((1+t)^{60} - 1)/t$ ,

Si on veut avoir tout remboursé au bout de 60 mois, il faut résoudre :

$10000(1+t)^{60} - 187 * ((1+t)^{60} - 1)/t = 0$ .

On tape :

```
fsolve(10000*(1+t)^60-187*((1+t)^60-1)/t,t=0.0033)
```

On obtient : 0.00385432769081

Si on fait 60 versements de  $r = 187$  euros, on aura payer en tout  $187 * 60 = 11220$  euros donc un peu moins que précédemment...mais le taux annuel est plus élevé ( $0.00385432769081 * 12 \simeq 4.62/100$ ). On est donc perdant...

## 6.6 Solution approchée de $f(x) = 0$ sur $[a; b]$

On va procéder par dichotomie : on partage l'intervalle  $[a; b]$  en deux :

$$[a, b] = [a; m = (a + b)/2] \cup [m = (a + b)/2; b]$$

et on regarde si  $f(a) * f(m) < 0$  si c'est le cas on cherche la solution dans  $[a; m]$  sinon on cherche la solution dans  $[m; b]$ . On recommence autant de fois qu'il faut pour avoir la précision  $eps$  désirée.

On tape :

```
Zero(f, a, b, eps) := {
  local m;
  si f(a)==0 alors return a fsi;
  si f(b)==0 alors return b fsi;
  si f(a)*f(b)>0 alors return "erreur" fsi;
  tantque (abs(a-b)>eps) faire
    m:=(a+b)/2;
    si (f(a)*f(m)<0) alors
      b:=m;
    sinon
      a:=m;
  fsi;
  ftantque;
  si (a<b) return evalf(a,b); sinon return evalf(b,a);
};;
```

Trouver un solution encadrement à  $10^{-9}$  près de  $x^2 - 2 = 0$  sur  $[0; 2]$ .

On tape :

$f(x) := x^2 - 2;$

`Zero(f, 0, 2, 10^-9)`

On obtient l'encadrement : 1.41421356145, 1.41421356238

## 6.7 Codage de messages

### 6.7.1 Les caractères utilisables

Les caractères utilisables dans `Xcas` ont un code ASCII qui va de 32 (l'espace) à 126 (le tilde), puis il y a des caractères spéciaux et les lettres accentuées qui ont un code ASCII qui va de 161 (le point d'exclamation à l'envers) à 255 (le bécarre). Avec `Xcas`, on utilise les commandes :

- `asc` qui a comme argument une chaîne de caractères renvoie la liste des codes ASCII. Par exemple `asc("AH@AH") = [65, 72, 64, 65, 72]`
- `char` qui a une liste de nombres renvoie la chaîne de caractères de code ASCII les nombres modulo 256. Ainsi `char([65+256, 72, 64, 65, 328+256]) = "AH@AH"`

### 6.7.2 Codage par addition

On peut convenir dans un premier temps de coder le message avec 27 caractères qui sont @, A,B,...Z et qui ont un code ASCII entre 64 et 90. On utilisera le caractère @ pour séparer les mots. Pour coder par addition, on choisit une clé secrète qui est ici un nombre  $c$  entre 0 et  $(126-90=36)$ . Pour coder un message on remplace la lettre de code  $n$  par la lettre dont le code est  $n + c$ . Ainsi si la clé vaut 7, le "A" qui a comme code 65 sera remplacé par la lettre de code  $65+7=72$  c'est à dire par le "H" et si la clé vaut 27, le "A" qui a comme code 65 sera remplacé par la lettre de code  $65+27=92$  c'est à dire par le "\". Dans ce cas le message codé utilise les caractères dont les codes vont de  $64 + c$  à  $90 + c < 127$ .

Pour décoder il suffit de remplacer  $c$  par  $-c$ .

On tape :

```
Codadd(L, cle) := {
  local s, j;
  s:=size(L)
  return char(asc(L)+ makelist(cle, 1, s));
};
```

On tape : `M1 :=Codadd("MESSAGE@SECRET@POUR@ZORRO", 7)`

On obtient :

```
"TLZZHNLGZLJYL [GWV\YGaVYYV"
```

On tape : `Codadd(M1, -7)`

On obtient : `"MESSAGE@SECRET@POUR@ZORRO"`

On tape : `M2 :=Codadd("MESSAGE@SECRET@POUR@ZORRO", 27)`

On obtient :

```
" "h`nn\b`[n`^m`o[kjpm[u]mmj" "
```

On tape : `Codadd(M2, -27)`

On obtient : `"MESSAGE@SECRET@POUR@ZORRO"`

On peut dans un deuxième temps vouloir coder le message uniquement avec des lettres de code compris entre 64 et 90 (pour que ce soit lisible). Pour cela on ramène

les codes entre 0 et 26 (en enlevant 64) puis on ajoute la clé, puis on ramène les nombres obtenus entre 0 et 26 en les remplaçant par le reste de la division par 27 et enfin on rajoute 64 pour avoir un code entre 64 et 90.

On tape :

```
Codaddi(L,cle) := {
  local s, j, M;
  s := size(L)
  M := asc(L) + makelist(-64, 1, s);
  pour j de 0 jusque s-1 faire
    M[j] := irem(M[j] + cle, 27) + 64;
  fpour
  return char(M);
};;
```

On tape : M2 := Codaddi ("MESSAGE@SECRET@POUR@ZORRO", 7)

On obtient :

"TLZZHNLGZLJYL@GWVAYGFVYYV"

On tape : Codadd(M3, -7)

On obtient : "MESSAGE@SECRET@POUR@ZORRO"

### 6.7.3 Codage par multiplication

On suppose que l'on n'utilise que les 27 caractères "A".."Z" et le caractère @ comme espace.

On tape : asc("@ABZ")

On obtient : [64, 65, 66, 90] Ainsi en enlevant 64 à ces codes on a des nombres entre 0 et 26.

On multiplie ces codes par le clé qui doit être un nombre inversible de  $Z/27Z$  par exemple si la clé vaut 7, on a  $7 * 4 = 28 = 1 \pmod{27}$ .

Les nombres inversibles de  $Z/27Z$  sont les nombres qui ne sont pas divisibles par 3. Pour avoir l'inverse d'un tel nombre il faut faire faire à Xcas les calculs dans  $Z/27Z$ .

On tape : 1/7 % 27

On obtient 4 % 27 ( $4 * 7 = 27 + 1 = 1 \pmod{27}$ )

On tape : 1/17 % 27

On obtient 8 % 27 ( $8 * 17 = 5 * 27 + 1 = 1 \pmod{27}$ )

C'est l'inverse de la clé que l'on utilisera pour le décodage.

On tape :

```
Codmult(L,cle) := {
  local s, M, j;
  s := size(L);
  M := asc(L) + makelist(-64, 1, s);
  pour j de 0 jusque s-1 faire
    M[j] := irem(M[j] * cle, 27) + 64;
  fpour;
  return char(M);
};;
```

On tape : `M4 :=Codmult ("MESSAGE@SECRET@POUR@ZORRO", 7)`

On obtient : `"JHYGVH@YHURHE@DXLR@TXRRX"`

On tape : `Codmult (M4, 4)`

On obtient : `"MESSAGE@SECRET@POUR@ZORRO"`

On remarque que le caractère @ reste inchangé car son code est ramené à 0 puis reste 0 après la multiplication..... Cela n'est pas forcément un problème puisqu'il code l'espace mais on peut connaître la longueur des mots.

On peut alors modifier le programme en enlevant 63 (pour éviter d'avoir 0) et en prenant le reste de la division par 28. Dans ce cas, c'est "?" de code 63 qui ne sera pas changé par le codage.

On tape :

```
Codmulti(L,cle):={
  local s,M,j;
  s:=size(L);
  M:=asc(L)+makelist(-63,1,s);
  pour j de 0 jusque s-1 faire
    M[j]:=irem(M[j]*cle,28)+63;
  fpour;
  return char(M);
};;
```

On tape : `1/9 % 28`

On obtient `-3 % 28`

On tape : `M5 :=Codmulti ("MESSAGE@SECRET@POUR@ZORRO", 9)`

On obtient : `"MYKKQOYHKYGBYTHLCABHRCBBC"`

On tape : `Codmulti (M5, -3)`

On obtient : `"MESSAGE@SECRET@POUR@ZORRO"`

Mais le plus simple est de travailler dans  $Z/29Z$  car 29 est premier et dans  $Z/29Z$  tous les nombres non nuls sont inversibles. On pourra avoir dans le message codé le caractère "?" de code 63.

On tape : `1/9 % 28`

On obtient `13 % 28.`

On tape :

```
Codmultip(L,cle):={
  local s,M,j;
  s:=size(L);
  M:=asc(L)+makelist(-62,1,s);
  pour j de 0 jusque s-1 faire
    M[j]:=irem(M[j]*cle,29)+62;
  fpour;
  return char(M);
};;
```

On tape `1/9 % 29`

On obtient `13 % 29`

On tape : `M6 :=Codmultip ("MESSAGE@SECRET@POUR@ZORRO@KKKK", 9)`

On obtient : `"QCMYUCPMCNDVPOFBDFDDDFP ? ? ? ?"`

On tape : `Codmultip (M6, 13)`

On obtient : `"MESSAGE@SECRET@POUR@ZORRO@KKKK"`

### 6.7.4 Codage sans utiliser `asc` et `char`

On peut aussi sans avoir besoin des fonctions `asc` et `char` écrire un fonction `Code` qui codera 29 caractères par exemple ">" par 0, "?" par 1, l'espace par 2, "A" par 3... "Z" par 28 et une fonction `Caract` qui renverra le caractère de code donné en argument.

On tape :

```
Code(a) := {
  local alphab, code, j;
  alphab:=">? ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  code:=table(seq(alphab[j]=j, j=0..28));
  return code[a];
}
;
```

On remarquera que l'on utilise ici un tableau `code` qui est indicé par des caractères : c'est ce que l'on appelle dans *Xcas* une *table*.

```
Caract(n) := {
  local alphab, code;
  alphab:=">? ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  return alphab[n];
}
;
```

On écrit le codage par addition en utilisant les fonctions `Code` et `Caract` :

```
Codadditi(L, cle) := {
  local s, j, M;
  s:=size(L)
  M:="";
  pour j de 0 jusque s-1 faire
  M:=M+Caract(alphab[irem(Code(L[j])+cle, 29)]);
  fpour
  return M;
}
;
```

Ou on écrit directement :

```
Codaddition(L, cle) := {
  local s, j, M, code, alphab;
  s:=size(L)
  alphab:=">? ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  code:=table(seq(alphab[j]=j, j=0..28));
  M:="";
  pour j de 0 jusque s-1 faire
  M:=M+alphab[irem(code[L[j]]+cle, 29)];

  fpour
  return M;
}
```

**On tape :** M7 :=Codaddition("MESSAGE SECRET POUR ZORRO",7)

**On obtient :**

"TLZZHNLGZLJYL>GWV?YGDVYYV"

**On tape :** Codaddition(M7,-7)

**On obtient :** "MESSAGE SECRET POUR ZORRO"

On écrit le codage par multiplication en utilisant les fonctions Code et Caract :

```
Codmultipli(L,cle):={
local s,j,M;
s:=size(L)
M:="";
pour j de 0 jusque s-1 faire
M:=M+Caract(irem(Code(L[j])*cle,29));
fpour
return M;
}
```

Ou on écrit directement :

```
Codmultiplica(L,cle):={
local s,j,M,alphab,code;
s:=size(L)
alphab:=">? ABCDEFGHIJKLMNOPQRSTUVWXYZ";
code:=table(seq(alphab[j]=j,j=0..28));
M:="";
pour j de 0 jusque s-1 faire
M:=M+alphab[irem(code[L[j])*cle,29)];
fpour
return M;
}
```

**On tape :** M8 :=Codmultiplica("MESSAGE SECRET POUR ZORRO KKKK",9)

**On obtient :**

"QCMMYUCPMCNDVPOFBDPRFDDFP????"

**On tape :** Codmultiplica(M8,13)

**On obtient :**

"MESSAGE SECRET POUR ZORRO KKKK"