Implémentation d'algorithmes sur Xcas en arithmétique

Rappels :
Il faudra structurer votre console, pour que le fichier sauvegardé soit lisible. N'oubliez pas de sauvegarder régulièrement votre fichier.
Pour différencier chaque exercice dans votre console Xcas, vous pouvez créer des groupements pour chaque exercice : menu `Edit --> Nouveau groupe`, avec possibilité de nommer le groupe obtenu.
Vous pouvez ensuite entrer des lignes de commentaires, qui ne seront pas exécutés et s'afficheront en vert : menu `CAS --> Nouveau commentaire`.
Vous pouvez réagencer également vos lignes de calculs, et en supprimer, en les sélectionnant au niveau de leur numéro, et en les déplaçant avec la souris, et en allant dans le menu `Edit --> Supprimer niveaux sélectionnés`.

Pour débuter

Exercice 0. *Premiers pas en programmation*
a. Pour éditer un programme allez dans le menu `Prg --> Nouveau programme`. Une fenêtre spécifique s'ouvre dans la ligne de la session. Un programme sur Xcas a toujours une structure analogue à la suivante :

```
f(a,b):={
 local j,k;
 instruction1;
 instruction2;
 retourne valeur;
}:;
```

L'exemple ci-dessus concerne un programme/une fonction qui a besoin de deux variables d'entrée appelées a et b. On peut bien sûr en utiliser plus, ou moins, ou pas du tout ; dans ce dernier cas, la première ligne est du type `Programme() :={` .
On remarquera que chaque ligne doit se terminer par un point-virgule `;`
Une fois le programme écrit il suffit de le compiler en cliquant sur le bouton `OK (F9)`. On peut ensuite l'exécuter dans les lignes de console qui suivent.
Il est conseillé de déclarer en début de programme les variables utilisées seulement à l'intérieur du programme (variables locales) par l'instruction `local var ;`. ATTENTION : ne pas utiliser `i` comme nom de variable, cette lettre étant réservée pour la constante mathématique i (racine carrée complexe de -1).

b. Un premier programme, calculant la somme des n premiers entiers :

```
som(n):={
 local j,s;
 s:=0;                      //initialisation de la variable contenant la somme
 pour j de 1 jusque n faire   // boucle pour
 s:=s+j;
 fpour;
retourne s;
}:;
```

On remarquera que l'affectation des variables se fait à l'aide du symbole `:=`, que l'on peut inclure des commentaires grâce à la commande //, et que l'on a utilisé la boucle "pour" dont la syntaxe est :
`pour var de vmin jusque vmax faire intructions ; fpour ;`

c. Utilisons maintenant l'instruction conditionnelle :
`si condition alors instruction1 ; sinon instruction2 ; fsi ;`
Pour cela écrivons un programme qui nous dit si un entier n divise l'entier m.

```
divise(n,m):={
 si irem(m,n)==0 alors
   afficher("n divise m");
 sinon
 afficher("n ne divise pas m");
 fsi;
}:;
```

Remarquons que le test d'égalité se fait à l'aide du double symbole `==` et non pas `=`.

d. Sauvegarde des programmes :
En plus de la sauvegarde des sessions (fichiers d'extension .xws), nous pouvons sauvegarder les programmes. Allez dans le menu `Prog` se trouvant dans la ligne de session du programme, et choisissez `Sauver` ou `Sauver comme`. Le fichier a alors une extension .cxx, et vous pouvez bien sûr l'importer ultérieurement grâce à la commande `Charger`.

Exercice 1. *Premier algorithme : liste des diviseurs d'un entier*

Nous allons implémenter l'analogue de la fonction `idivis()`, qui renvoie la liste des diviseurs (entiers positifs) d'un entier n.
Concernant la manipulation des listes, nous allons utiliser la fonction `append(l,a)` qui ajoute l'élément a à la liste l ; d'autre part, on peut créer une liste vide l par la commande `l :=[] ;`
Et bien sûr, nous pouvons nous servir de ce qui a été vu dans l'exercice 0 : boucle pour, test conditionnel de divisibilité d'un entier par un autre.
Traduire cet algorithme en langage Xcas donnant la liste des diviseurs (positifs) d'un entier naturel :

```
Programme listediviseur(n)
 Variables: l (liste), j (entier)
 l:=liste vide (initialisation liste l)
 Pour j allant de 1 à n
  Si j divise n alors ajouter j à la liste l
  Finsi
 Finpour
 Retourner l
```

Pour aller plus loin : Vous pouvez ensuite améliorer votre programme pour qu'il puisse fonctionner avec un entier relatif, pour qu'il soit plus rapide (par exemple en remarquant que si j divise n alors n/j divise aussi n), pour qu'il compte le nombre de diviseurs d'un entier, pour qu'il teste si un nombre est premier ou non.

Exercices à rendre : PGCD

Exercice 2. *Algorithme d'Euclide : calcul de pgcd*

On rappelle que l'algorithme d'Euclide est basé sur le principe suivant. Soit $(a, b) \in \mathbb{N} \times \mathbb{N}^*$. Posons $r_0 = a$, $r_1 = b$. On définit une suite finie d'entiers (r_i) par : pour $i \geqslant 1$, si $r_i \neq 0$, r_{i+1} est le reste de la division

euclidienne de r_{i-1} par r_i, sinon la suite s'arrête. Alors le PGCD de a et b est égal au dernier élément non nul r_n de cette suite.
Traduisez en langage Xcas l'algorithme d'Euclide simple de calcul de pgcd de deux entiers naturels. Il est conseillé d'utiliser la boucle tant que dont la syntaxe est :
`tant que condition faire instructions ; ftantque ;`
D'autre part, pour tester si un nombre a est non nul il faut utiliser la syntaxe `a !=0`.
Rappel : le reste de la division euclidienne de a par b est `irem(a,b)`.

Exercice 3. *Algorithme d'Euclide étendu : calcul de PGCD et de coefficients de Bezout*

On rappelle que l'algorithme d'Euclide étendu est basé sur le principe suivant. On garde les notations de l'exercice 2, en introduisant les quotients q_i des divisions euclidiennes : $r_{i-1} = r_i q_{i+1} + r_{i+1}$. On définit deux suites finies (u_i) et (v_i) par $(u_0, u_1) = (1, 0)$, $(v_0, v_1) = (0, 1)$, et pour $1 \leqslant i \leqslant n-1$, $u_{i+1} = u_{i-1} - u_i q_{i+1}$ et $v_{i+1} = v_{i-1} - v_i q_{i+1}$.
Alors, on montre que pour tout i, $r_i = au_i + bv_i$ et en particulier $r_n = pgcd(a, b) = au_n + bv_n$.
Traduisez en langage Xcas l'algorithme d'Euclide étendu de calcul de pgcd de deux entiers naturels, et donnant de plus les entiers relatifs u et v tels que $au + bv = pgcd(a, b)$ (relation de Bezout).

ATTENTION : il faut bien prévoir et gérer toutes les variables nécessaires à l'exécution de l'algorithme.

Pour aller plus loin : Vous pouvez utiliser des variables sous forme de liste et faire des opérations (additions, multiplications) directement sur ces listes.

Exercice 4. *Algorithme d'Euclide : compléments*

a. Testons l'efficacité en terme de complexité algorithmique (et donc de temps de calcul) de l'algorithme d'Euclide. Xcas possède la fonction `time(fonction)` permettant de donner le temps d'exécution d'une fonction (ou d'un programme).
Par exemple `time(gcd(a,b))` donne le temps d'exécution du calcul de pgcd implémenté dans Xcas.
Pour un calcul de pgcd de tête avec des petits entiers, on préfère en général utiliser la factorisation en facteurs premiers des deux entiers. On en déduit facilement le pgcd. Comparer alors les temps d'exécution du calcul de pgcd par l'algorithme d'Euclide (votre programme ou celui de Xcas), avec le temps d'exécution de la factorisation de deux entiers. N'hésitez pas à prendre de très grands entiers !!

b. Allons plus loin dans l'étude de la complexité algorithmique de l'algorithme d'Euclide.
Modifiez votre programme initial de calcul de pgcd élaboré dans l'exercice 2 pour qu'il puisse vous donner le nombre de divisions euclidiennes effectuées en tout pour un couple (a, b) donné (indice : nombre de passages dans la boucle pour).
Implémentez ensuite l'algorithme suivant, permettant, à partir d'un entier N fixé, de déterminer le couple (a, b), vérifiant $a \leqslant b \leqslant N$, et maximisant le nombre de divisions euclidiennes effectuées pour le calcul de $pgcd(a, b)$.

```
Programme couple_max(N)
 Variables: s, a, b, j, k   (entiers)
 s:=0; a:=0; b:=0;           (initialisation variables)
 Pour k allant de 1 à N
      Pour j allant de 1 jusque k
         t:=nb_divisions_pgcd(j,k)   (calcul du nombre de divisons à effectuer )
         Si t>s Alors
          s:=t; a:=j; b:=k;  (on a trouvé un couple avec un plus grand nombre de divisions)
         FinSi
      Finpour
 Finpour
 Retourner s et (a,b)        (on a trouvé notre couple maximal)
```

Faire tourner votre programme, pour des valeurs de N inférieures ou égales à 1000 (au-delà, le temps de calcul est un peu long).
Que pensez-vous des couples retournés ? Ne vous font-ils pas penser à une suite de nombres entiers très célèbre ? En déduire une conjecture sur le couple d'entiers inférieurs à N et maximisant le nombre de divisions euclidiennes effectuées par l'algorithme d'Euclide.
A l'aide de l'expression en fonction de n des termes de cette célèbre suite, en déduire l'ordre de grandeur du nombre maximal de divisions euclidiennes effectuées lorsque l'on exécute l'algorithme d'Euclide pour un couple d'entiers inférieurs à N. Comparer au théorème de Lamé (cherchez sur internet).

c. Pour aller plus loin, avec l'algorithme d'Euclide.
Vous pouvez améliorer votre programme pour qu'il puisse calculer le pgcd de r entiers (par la propriété $pgcd(a1,..,ar) = pgcd(a1, pgcd(a2,...,ar))$), ou le modifier complètement en utilisant un algorithme récursif pour calculer le pgcd de deux entiers.

Pour ceux qui ont envie d'aller plus loin

Exercice 5. *Algorithme rapide de calcul d'une puissance modulaire*

Nous allons voir ce qui se cache derrière la commande `powmod` de Xcas.

L'idée de l'algorithme est la suivante. Si vous souhaitez calculer a^{13}, l'algorithme naïf consistant à exécuter une boucle va effectuer 12 multiplications. Or, en remarquant que $a^{13} = a \cdot ((a \cdot a^2)^2)^2$, nous n'utilisons plus que 2 multiplications, et 3 élévations au carré, soit 5 multiplications en tout.
On démontre plus généralement, que si la décomposition de n en base 2 est $n = 2^k + n_{k-1} \cdot 2^{k-1} + \cdots + n_1 \cdot 2 + n_0$ avec $n_i = 0$ ou 1, alors : $a^n = a^{n_0}(a^{n_1}(a^{n_2}(\cdots(a^{n_{k-1}} \cdot a^2)^2)\cdots)^2)^2$. Nous effectuons donc au plus k multiplications et k élévations au carré.
Cette idée peut ainsi s'utiliser pour différents calculs de puissances entières : puissances d'un entier, d'un réel, d'une matrice, d'un polynôme, ou dans notre cas, d'un entier modulaire.

A vous d'implémenter cet algorithme. Pour décomposer n en base 2, vous pouvez écrire un programme, ou bien utiliser directement la fonction implémentée sur Xcas, appelée `convertir(n,base,b)` (elle convertit l'entier n selon la base b).

Exercice 6. *Algorithme binaire de calcul de pgcd*

C'est un algorithme récursif, qui n'utilise que des multiplications ou des divisions par 2, ce qui devient très efficace quand les nombres entrés sont écrits en base 2.

Voici l'algorithme (écrit ici pour des nombres en base quelconque, et donc en particulier en base 10) :

```
Programme pgcd_binaire(a,b)
  Si a=b Alors Retourner a
  Si a et b sont pairs (et a différent de b) Alors Retourner 2*pgcd_binaire(a/2,b/2)
  Si a est pair (mais pas b) Alors Retourner pgcd_binaire(a/2,b)
  Si b est pair (mais pas a) Alors Retourner pgcd_binaire(a,b/2)
  Si a et b sont impairs et a>b Alors Retourner pgcd_binaire((a-b)/2,b)
  Si a et b sont impairs et b>a Alors Retourner pgcd_binaire((b-a)/2,a)
```

On remarquera que la liste des cas possible est bien exhaustive.
Implémentez cet algorithme dans Xcas.