بِسمِ اللَّهِ الرَّحمَنِ الرَّحِيمِ
وَ عَلَى اللَّهِ فَليَتَوَكَّلِ المُتَوَكِّلون

صَدَقَ اللَّهُ العَظِيم

Programmation Maple© : *Initiation sur des exemple simples*

1 Annonce

Dans le cadre des activités organisées par le BDE des CPGE Med V Casablanca, et dans le but d'intéresser au mieux les élèves aux journées d'informatiques prevues aux CPGE Med V, Casablanca les 22-23 Mars, Mr Mamouni professeur de mathématiques en classes MPSI à le plaisir dinviter les élèves des CPGE Med V, et toute autre personne interessée à deux séances dinitiation à la programmation et l'algorithmique.

1) **1ère séance** : vendredi 7 Mars 2008, 16h30-18h :
   a) Intitulé : Initiation sur des exemples simples
   b) Contenu :
      - Cours de programmation
      - Tester si un nombre est premier
      - Décomposer un entier en base b
      - Methode de dichotomie
      - Calcul approché de π
      - Cryptographie RSA
2) **2ème séance** : Vendredi 14 Mars 2008, 16h30-18h :
   a) Intitulé : Programmation avancéé.
   b) Contenu :
      - Corrigé de l'épreuve d'informatique, Centrale-Sup Elec, 2004.
      - Corrigé de l'épreuve d'informatique, X-Polytechnique, 2003.

Language de programmation : Maple ©

Mamouni My Ismail
Professeur de mathématiques
MPSI 2, CPGE Med V
Casablanca.

2 Cours résumé.

1) Objects manipulés
- Objects élémentaires : numériques et caractères.
- Objects structurés : chaines de caractères, tableaux,..

2) Actions utilisés.
- Affectation
- Appel d'un algorithme prédefini.

3) Instructions de contrôle.

a) La boucle for.
On lutilise pour exécuter des instructions dépendant dun indice entier ou pour exécuter une même instruction n fois.
Syntaxe :
for k from valeur initiale de k to valeur finale de k by le pas
do instructions ;
od ;
Exemple : Algorithme d'Euclide, les restes successives de la division euclidienne.

```
> a:=15487:b:=1254:liste:=NULL:
> from 1 to 5 do
> r:=irem(a,b):
> liste:=liste,r:
> a:=b:
> b:=r:
> od:
> [liste];
```

$$[439, 376, 63, 61, 2]$$

Ce petit programme présente un défaut à chaque fois qu'on veut l'appliquer pour de nouvelles valeurs de $a, b, n = 5$ on doit recharger le programme. Pour y rémedier on va le nommer, lui associer des variables, ici a, b, n, appelons le "restes", le syntaxe en Maple©est le suivant :

```
> restes:=proc(c,d,n)
> a:=c:b:=d:liste:=NULL:
> from 1 to n do
> r:=irem(a,b):
> liste:=liste,r:
> a:=b:
> b:=r:
> od:
> RETURN([liste]);
> end:
```

```
\begin{Maple Warning}{\normalsize{Warning, `a` is implicitly declared local to procedu
```

```
\begin{Maple Warning}{\normalsize{Warning, `b` is implicitly declared local to procedu

\begin{Maple Warning}{\normalsize{Warning, `liste` is implicitly declared local to pro

\begin{Maple Warning}{\normalsize{Warning, `r` is implicitly declared local to procedu

\begin{Maple Warning}{\normalsize{Warning, `a` is implicitly declared local to procedu

\begin{Maple Warning}{\normalsize{Warning, `b` is implicitly declared local to procedu

\begin{Maple Warning}{\normalsize{Warning, `liste` is implicitly declared local to pro
```

Remarque : Maple©déclare automatiquement toutes les variables locales quand on oublie de le faire.
Exemple numérique :

```
> restes(15487,1254,6);
```

$$[439, 376, 63, 61, 2, 1]$$

Remarque : Ce programme présente à son tour un autre défaut, les restes successives ne sont plus possibles dés que l'un des reste est nul.
Exemple numérique :

```
> restes(15487,1254,8);
```

```
\begin{Maple Error}{\normalsize{Error, (in restes) numeric exception: division by zero
```

Ce qui nous amène une autre solution : "test d'arrêt"

b) Le test d'arrét while.
Syntaxe : while le test d'arrêt n'est pas remplie do instructions od :
Exemple : Algorithme d'Euclide.

```
> euclide:=proc(c,d) local a,b,r,Restes:
> a:=c:b:=d:
> r:=irem(a,b):Restes:=r:
> while r>0 do
> a:=b:b:=r:r:=irem(a,b):Restes:=Restes,r:
> od:
> [Restes];
> end:
```

Remarque : Ici on a déclaré nous même nous variables comme étant locales.
Application numérique :

```
> c:=156989:d:=165:euclide(c,d);
```

$$[74, 17, 6, 5, 1, 0]$$

Remarque : Le dernier reste non nul dans l'algorithme d'Euclide n'est autre que le pgcd, chose qu'on peut vérifier rapidement avec le programme prédéfini en Maple©appelé igcd.
Application numérique :

```
> igcd(c,d);
```

c) Les branchements if.
Syntaxe : If condition réalisé then instructions else instructions fi ;
Exemple :Méthode de dichotomie
Objectif :Localiser une racine de l'équation $f(x) = 0$.
Principe : Si $f(a)f(b) < 0$ alors $f(x) = 0$ admet une solution entre $[a, b]$. On pose $c = \dfrac{a+b}{2}$, si $f(a)f(c) < 0$ alors la solution dans $[a, c]$, sinon elle est dans $[c, b]$.
Application numérique :

```
> f:=x->x^2-2*x+1:
> a:=0:b:=3/2:c:=(a+b)/2:
> if f(a)*f(c)<0 then b:=(a+b)/2
> else a:=(a+b)/2
> fi:
> [a,b];
```

$$[3/4, 3/2]$$

3 Initiation sur des exemples simples.

3.1 Tester de primalité.

Objectif. Ecrire un programme qui verifie si un nombre est premier à l'aide du crible d'Eratosténe, tou d'abord on écrit un programme qui donne tous les nombres premiers inferieurs a un entier donné.

```
> liste:=proc(n)local a, list:
> a:=prevprime(n):
> list:=a:
> while a<>2 do
> a:=prevprime(a):
> list:=a,list:
> od:
> [list]:
> end:
> liste(54);
```

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]$$

```
> test:=proc(n)local A,i,a:
> A:=liste(trunc(sqrt(n)));
> for i from 1 to nops(A) do
> a:=irem(n,op(i,A)):
> if a<>0 then next
> else break
> fi:
> od:
```

```
> if i<nops(A) then print('n-non-premier'):
> else print('n-premier'):
> fi:
> end:
> test(102);
```

$$102 - non - premier$$

```
> test(101);
```

$$101 - premier$$

3.2 Numération en base b.

Objectif : Écrire un nombre en base b.
Principe : On sait que la décomposition d'un entier naturel en base b quelconque repose sur les restes successifs de sa division euclidienne par b.
Programme :

```
> decomposer:=proc(n,b) local r,q,liste,p:
> r:=irem(n,b):q:=iquo(n,b):liste:=r:
> while q>=b do r:=irem(q,b):p:=iquo(q,b):q:=p:liste:=r,liste:od:[q,liste];end:
> decomposer(159871,10);
```

$$[1, 5, 9, 8, 7, 1]$$

```
> decomposer(7,2);
```

$$[1, 1, 1]$$

3.3 Calcul approché de π avec la méthode des isopêrimétres

. **Principe**.

1) Soit $(a, b) \in (\mathbb{R}^{*+})^2$ tel que $a < b$, on pose :

$$\begin{cases} a_0 = a & , b_0 = b \\ a_{n+1} = \dfrac{a_n + b_n}{2} & , b_{n+1} = \sqrt{a_{n+1} b_n} \end{cases}$$

Montrer que ces suites sont bien définies et adjacentes.

2) Exprimer leurs limites communes en fonction de $\theta = \arccos\left(\dfrac{a}{b}\right)$.
Indication : On pourra d'abord commencer par exprimer les a_n, b_n en fonction de θ.

3) soit $n \in \mathbb{N}$, et P_n le polygône régulier à 2^n côtés et de périmètre 2, soit r_n le rayon du cercle inscrit et R_n celui du cercle circonscrit à P_n, montrer que $\forall n \geq 2$ on a :

$$r_{n+1} = \frac{r_n + R_n}{2}, R_{n+1} = \sqrt{r_{n+1} R_n}$$

4) En déduire qu'elle sont adjacentes puis en remarquant que :
$\dfrac{1}{R_n} \leq \pi \leq \dfrac{1}{r_n}$ en déduire que $\dfrac{1}{R_n}, \dfrac{1}{r_n}$ convergent vers π.

Programme.

```
> n := 0 : R := 1/sqrt(8) : r := 1/4 : N := 10 :
> r1:=1/r:R1:=1/R:
> epsilon:=evalf(10^(-N)):
> erreur:=evalf(abs(r1-R1)):
> while erreur>evalf(epsilon) do
> r:=evalf((r+R)/2);
> R:=evalf(sqrt(R*r));
> r1:=1/r;R1:=1/R;
> erreur:=evalf(abs(r1-R1)):n:=n+1:
> od:
> print(‘la valeur approchée par defaut est‘=evalf(r1));
        ‘la valeur approchée par defaut est‘ = 3.141592655
> print(‘la valeur approchée par exces est‘=evalf(R1));
         ‘la valeur approchée par exces est‘ = 3.141592655
> print(‘le nombre d’operation‘=n);
                    ‘le nombre d’operation‘ = 17
```

3.4 Cryptographie RSA.

Principe. Soit p et q deux nombres premiers, on pose $n = pq$. Soit M un entier naturel premier avec pq, qui représente le message à décoder, et C le message codé envoyé.

1) Dites pourquoi $\varphi(n) = (p-1)(q-1)$.

2) Soit e premier avec $\varphi(n)$, justifier l'existence de $d \in \mathbb{Z}$ tel que $ed \equiv 1 \pmod{\varphi(n)}$.

3) Le message M est codé en C tel que $C \equiv M^e \pmod{n}$.
En déduire que : $C^d \equiv M \pmod{n}$.

Indication : On pourra penser à utiliser le théorème d'Euler.
Remarque : Le couple (n, e) est appelé clef publique alors que le couple (n, d) est appelé clef privée. On constate que pour chiffrer un message, il suffit de connaitre e et n. En revanche pour déchiffrer, il faut d et n. Ainsi il suffit de connaitre p, q et e puisque $\varphi(n) = (p-1)(q-1)$ et $d \equiv e^{-1} \quad [\varphi(n)]$. **Programme.** ***On se donne d'abord des nombres premiers, plus qu'ils sont grand plus que c'est mieux, on comprendra dans la suite pourquoi.***

```
> p:=13:q:=17:
```

On factorise le produit (p-1)(q-1) pour en déduire un nombre premier avec.

```
> ifactor((p-1)*(q-1));
```

(0)

```
> e:=5:
```

On cherche les coéfficient de Bezout u,d tels que u(p-1)(q-1)+ed=1.

```
> igcdex((p-1)*(q-1),e,'u','d'):d;
```

77

On se donne le message à coder.

```
> Message:=162:
```

On code le message.

```
> Code:=Message^e mod p*q;
```

$$Code := 93$$

On décode le message codé.

```
> Decodage:=Code^d mod p*q;
```

$$Decodage := 162$$

Oh ça marche,... mais si on prend le message assez grand ?

```
> Message:=1452:
> Code:=Message^e mod p*q;
```

$$Code := 198$$

```
> Decodage:=Code^d mod p*q;
```

$$Decodage := 126$$

Oh ça ne marche plus, la raison c'est que le message dépasse pq, alors que le décodage non , parceque c'est son reste mod (pq), vérifions.

```
> Message mod p*q;
```

126

Pour y rémedier on découpera le message initial en blocs tous inférieurs à pq. On définit une fonction appellée, hachage qui extrait le 1ér bloc.

```
> hachage:=proc(M,p,q) local n,a,b;
> for n from 1 to nops(M)  do
> a:=sum(M[k]*10^(k-1),k=1..n):b:=a-M[n]*10^(n-1):
> if a>=p*q then break fi:
> od:return([b,n-1]):
> end proc:
```

Puis on définit une fonction appellée, décomposition qui extrait le 2èm bloc à partir du 1ér et ansi de suite jusqu'à extraire tous les blocs..

```
> decomposition:=proc(M,p,q) local M1,n,m,Blocs,M2;
> M1:=M;n:=0;m:=0;Blocs:=NULL;
> while m<nops(M) do
> M2:=op(1,hachage(M1,p,q)):
> Blocs:=M2,Blocs:
> n:=op(2,hachage(M1,p,q)):
> m:=m+n:
> M1:=[seq(M[i],i=m+1..nops(M))]:
> od:
> return(convert(Message,base,10)[1],Blocs);
> end proc:
```

Vérifions sur un exemple.

```
> p:=13;q:=17;Message:=50698465;M:=convert(Message,base,10):
```

$$p := 13$$
$$q := 17$$
$$Message := 50698465$$

```
> Blocs:=decomposition(M,p,q);
```

$$Blocs := 5, 0, 69, 84, 65$$

```
> Code:=seq([Blocs][i]^e mod p*q,i=1..nops([Blocs]));
```

$$Code := 31, 0, 205, 67, 182$$

```
> Decodage:=seq([Code][i]^d mod p*q,i=1..nops([Code]));
```

$$Decodage := 5, 0, 69, 84, 65$$

Ah Enfin ça marche, c'est joli la programmation non, pour terminer je vous laisse un exercice.
Ecrire un programme qui transforme les lettres en chiffres et un autre qui fait l'inverse.
C'est pas difficile, il suffit de s'y mettre.

Fin.