Maple TD2

1 Correction des exercices précédents

1.1 Factorielle

1. cf plus bas.
2.
```
fact1 := proc(n)
local x,i;
if n < 1 then return NON end if;
x:=1; for i from 1 to n do   x := x * i;  end do; return x;
end proc;
```
3.
```
fact2 := proc(n)
if n < 2 then return 1;
else return ( n * fact2(n-1));
end proc;
```
4. `ifactor` décompose son paramètre en facteurs premiers. Il suffit alors de constater qu'un 0 à la fin c'est un 10 dans la décomposition, soit $2 * 5$. Il faut donc compter l'exposant de 5 dans la décomposition en facteurs premiers

1.2 Fibonacci

1.
```
fib := proc(n)
if n < 2 then return 1;
else return fib(n-1) + fib(n-2);
end proc;
```
2. Le calcul de `fib(n)` devient de plus en plus long quand n augmente.
3. En faisant très attention quand on utilise cette construction...

   ```
   1; 1; % + %%; % + %%; % + %%; % + %%;
   ```
4. Cf cours de math.
5.
```
suites := proc(u0, u1, a, b)
 local res1,r1,r2,s,alpha,beta;
 res1 := solve(r^2-a*r-b=0,r);
 r1 := res1[1];r2 := res1[2];
 if r1 = r2 then
   s := solve( {u0=alpha+beta, u1=alpha*r1+beta*r1}, {alpha,beta} );
   assign(s); RETURN(alpha*r1 ^ n + n*beta*r1^n)
 else
   s := solve( {u0=alpha+beta, u1=alpha*r1+beta*r2}, {alpha,beta} );
   assign(s); RETURN(alpha*r1 ^ n + beta*r2^n)
 fi;
 end;
```
6. On montre que $T(n)$ suit la même loi de croissance que $fib(n)$, en effet, pour calculer $fib(n)$ on doit calculer $fib(n-1)$ et $fib(n-2)$ soit faire $T(n-1) + T(n-2)$ appels. On a donc $T(n) = T(n-1) + T(n-2)$.

2 Programmes et algorithmes

Vous savez déjà donner une instruction à Maple. Une instruction est une commande, terminée en général par un ; Ceci est, par exemple une instruction :

```
2*5;
```

On peut aussi mettre plusieurs instructions à la suite, pour faire des choses un peu plus compliquées.

Un algorithme est la donnée d'une série d'instructions décrivant comment résoudre un problème donné (par exemple, calculer $n!$). Cela s'apparente à une recette de cuisine : on décrit la liste des opérations à effectuer pour arriver au résultat, le résultat étant le plat cuisiné pour une recette, la valeur attendue pour un algorithme. Pour décrire un algorithme, on se limite à un certain nombre d'instruction, qui pouront être comprises par l'ordinateur. Toute la difficulté est de décomposer le problème que l'on veut résoudre en petites étapes, chacune devant être comprise par Maple, et leur succession devant aboutir au résultat. En plus des instructions que vous avez vu la dernière fois (appel de fonction, affectation, petits calculs...), Maple fournit divers outils pour définir comment enchaîner ces opérations :

1. Les procédures, permettent de regrouper des instructions en fonction d'un ou plusieurs paramètres, pour ensuite pouvoir rappeler ce code autant de fois que nécessaire. Par exemple

   ```
   foisdeuxplusun := proc(n) return 2*n+1; end proc;
   ```

2. Les boucles `for`, permettent de répéter une série d'instructions un nombre fixé de fois, en faisant varier un indice indiquant le nombre de fois où on est passé par la boucle. Par exemple :

   ```
   for i from 1 to n do
     print(i);
     print(foisdeuxplusun(i));
   end do;
   ```

   Le code entre do et end do est répété pour chaque i compris entre 1 et n

3. La structure `if condition then faireqqch else faireautrechose end if ;` vous permet de tester une condition de d'exécuter une instruction différente si elle est vraie ou fausse. Par exemple

   ```
   valeurabsolue := proc(x)
   if x >= 0 then
     return x;
   else
     return -x;
   end if;
   end proc;
   ```

4. La boucle `while` est un mélange de if et de for. Elle s'écrit `while condition do code end do` et le code est répété tant que condition est vraie. Par exemple :

   ```
   n:= 12;
   while n > 0 do
     print(n);
     n := n - 1;
   end do;
   ```

   Va afficher les entiers de 12 à 1.

Il est possible de combiner ces constructions :

```
valeurabsolue := proc(x)
if x >= 1 then
  variable := 1
  for i from 1 to x do
    variable := variable*x
  end do
  return foisdeuxplusun(variable);
else
  print "NON !"
  return -1
end if;
end proc;
```

Quand vous écrivez un algorithme, il faut réfléchir à plusieurs questions :

1. Est-ce que mon algorithme calcule le bon résultat ?
2. Est-ce que j'ai un argument pour m'en convaincre ?
3. Est-ce que mon algorithme termine ? Si vous faites par exemple :

   ```
   x:= 1;
   while x > 0 do
     x := x + 1
   end do;
   ```

   Maple va se bloquer, ce calcul est infini. Pourquoi ?
4. Si mon algorithme doit terminer, pourquoi est-ce que c'est vrai ?
5. Est-ce que mon algorithme est efficace ? Vous avez vu qu'il y a plusieurs méthodes pour calculer la suite du Fibonacci, toutes ne sont pas aussi rapides, et parfois le calcul va être trop long.
6. Comment se comporte mon algorithme quand les entrées sont bizarres. foisdeuxplusun(foisdeuxplusun) ?

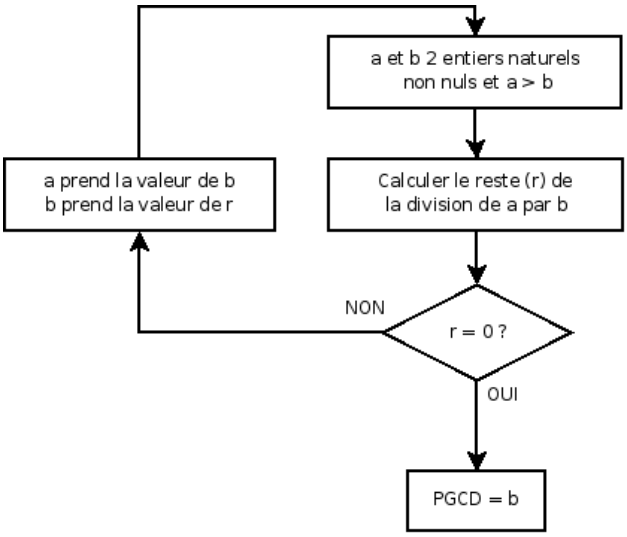
3 Exercices

3.1 Manipulations simples

1. Écrire deux procédures calculant la somme des n premiers nombres pairs une fois avec une boucle for et une fois avec une boucle while.
2. Écrire une procédure prenant en paramètre un réel x et un entier n et renvoyant x^n avec une boucle for, puis avec une boucle while, puis avec if then else et la récursivité. On rappelle que $x^n = x * x^{n-1}$.

3.2 PGCD

On rappelle l'algorithme d'Euclide pour le calcul du pgcd, sous forme d'un schéma :



Programmez le en maple, à l'aide d'une boucle `while`.

3.3 Séquences

En plus des entiers, des procédures ou des flottants, Maple a un type intégré appelé séquence. Il sert à regrouper des objets dans une sorte de liste. Par exemple `1,2,3,4` est une séquence. Vous pouvez accéder au i^{eme} élément d'une séquence à l'aide de `[i]`. Par exemple : `sequence :=0,1,2,3,4,5 ; sequence[3] ;`. Vous pouvez aussi mettre des séquences bout à bout, en faisant `sequence1,sequence2` et les modifier à l'aide de `subs`.

1. Étudier comment fonctionne les fonctions `seq`, `subs` et `nops`
2. Écrire une instruction renvoyant une séquence contenant les n premiers entiers pairs.
3. Écrire une procédure prenant une séquence en paramètre et renvoyant son plus grand élément.
4. Écrire une procédure prenant une séquence en paramètre et renvoyant la somme de ses éléments.
5. Écrire à nouveau la procédure du 2.1.1 avec les réponses aux questions précédentes.

3.4 Suite de Syracuse

Par définition, la suite de Syracuse est :

$$\begin{cases} u_0 \text{un entier strictement positif} \\ u_{n+1} = 3u_n + 1 \text{ si } u_n \text{ est impair} \\ u_{n+1} = \frac{u_n}{2} \text{ si } u_n \text{ est pair} \end{cases}$$

1. Écrire une fonction f prenant en argument u_n et renvoyant u_{n+1}
2. Tester la fonction f en calculant les 4 premiers termes de la suite définie par $u_0 = 4$.
3. Que se passe-t-il si à un moment $u_n = 1$?
4. On appelle orbite d'un entier positif u_0 la séquence formée par $u_0, u_1, ..., u_n$ avec n le premier indice pour lequel $u_n = 1$. Écrire une procédure prenant u_0 et donnant son orbite.
5. Il n'est pas évident que cette fonction termine, c'est en fait encore un problème ouvert. Écrire une procédure qui prend en paramètre n et vérifie pour $u_0 \in [1..n]$ calcule la taille maximale des orbites (utilisez la fonction *nops*).
6. Écrire une procédure effectuant le tracé d'un orbite en joignant les points (k, u_k). Utiliser les fonctions plot et seq.