UE Algorithmes et programmation
L1/S2 – TD/TP n^o 4
Fonctions – Année 2018

Vous pouvez récupérer du code ici :
`http://lab-sticc.univ-brest.fr/~rodin/FTP/Enseignements/L1/AlgoEtProg`

Sans indication particulière, les fonctions ne doivent pas être écrites sous forme récursives. La récursivité ne sera utilisée que lorsque cela sera indiqué explicitement dans le sujet.

TD

1 TD Exercice 0 : petites fonctions

1. Ecrire une fonction `trois` sans paramètre et retournant un `int`. La valeur retournée par cette fonction sera l'entier `3`.
2. Ecrire une fonction `plus` permettant d'augmenter la valeur d'une variable de type `int` d'une certaine quantité `val`.
3. Ecrire un programme principal utilisant ces deux fonctions.

2 TD Exercice 0' : petite fonction mathématique

1. Ecrire une fonction `div` qui, à partir de 2 entiers passés en paramètre, calcule le quotient et le reste de la division entière de `a` par `b`.

3 TD Exercice 1 : petites fonctions mathématiques

1. Ecrire une fonction `poly` permettant de calculer les racines d'une équation du second degré du type `a` x^2 `+ b` x `+ c = 0`

   On prendra soin de bien définir :
   - les paramètres d'entrée et de sortie de cette fonction,
   - les préconditions sur les paramètres d'entrée.

   Pour tester cette fonction, écrire un programme principal :
   - décrivant ses compétences,
   - demandant à l'utilisateur de saisir des valeurs pour `a`, `b` et `c`,
   - appelant la fonction `poly`,
   - affichant le nombre de racines et ces racines si elles existent.
2. Le but est ici d'écrire une fonction `mult` permettant de calculer la multiplication "égyptienne" de deux entiers `a` et `b` positifs ou nuls.

Cette fonction utilise uniquement comme opérations primitives : – la multiplication par 2, – la division par 2, – l'addition et – la soustraction de 1.
Exemple :

```
25 * 19 =    25 * 18 + 25
        =    50 * 9  + 25
        =   (50 * 8  + 50) + 25
        =   100 * 4  + 75
        =   200 * 2  + 75
        =   400 * 1  + 75
        = ((400 * 0) + 400) + 75
        =   475
```

Effectuer l'analyse de ce problème. Ecrire ensuite le code de cette fonction `mult` ainsi qu'un programme principal l'utilisant.

3. En utilisant la méthode des développements limités, écrire une fonction `myCos` permettant de calculer le $cos(x)$.
   Rappel :
   $$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} + ...$$
   On négligera les termes dont les valeurs absolues sont inférieures à une certaine valeur `epsilon` (par exemple `1.0e-4`).

4 TD Exercice 2 : PGCD et PPCM

4.1 Rappels :

1. Le PGCD (plus grand commun diviseur) de deux entiers positifs a et b est le plus grand nombre x tel que x divise a et b.
   L'algorithme d'Euclide, qui permet de calculer le PGCD de deux nombres a et b, vient de l'idée suivante :
   — si b = 0 alors le PGCD de a et b est a
   — sinon, alors le PGCD de a et b est égal au PGCD de b et a modulo b.
   Pour calculer le PGCD de a et b à l'aide de cette idée, on propose d'utiliser une boucle "tantque" , en modifiant les valeurs de a et b à chaque itération, jusqu'à avoir b = 0.
2. le PPCM (plus petit commun multiple) de a et b est égal à ab divisié par le PGCD de a et b.

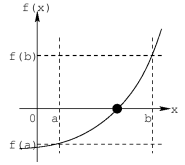
4.2 Exercices

1. Ecrire une fonction calculant le PGCD de 2 nombres a et b
2. Ecrire une fonction calculant le PPCM de 2 nombres a et b
3. Ecrire une fonction calculant le PGCD et le PPCM de 2 nombres a et b
4. Ecrire un programme principal utilisant toutes ces fonctions

5 TD Exercice 3 : recherche du zéro d'une fonction

S'il reste du temps...

Ecrire une fonction `zero` renvoyant le zéro d'une fonction f, le zéro se trouvant dans l'intervalle $]a,b[$.
La fonction `zero` cherchera donc un x tel que $f(x) = 0$ dans $]a,b[$ avec un seuil de détection `seuil` (par exemple, `seuil = 1.e-3`).
La fonction f est supposée continue sur $[a,b]$. De plus, la fonction f doit être telle que $f(a).f(b) < 0$. Ainsi, il existe une racine (supposée unique) de f dans $]a,b[$
La figure suivante présente une telle fonction.



On peut, par exemple, considérer la fonction f (ayant un zéro en `2.`) suivante :

```
double fct(double x)
{
 if (x < 0.0) { printf("Erreur argument de la fonction fct\n"); exit(1); }

 return x*x - 4;
}
```

Exemple d'appel de la fonction `zero` :

```
int main(void)
{
 ...
 double z;
 ...
 z = zero(0.2,10.3);   /* 1.999741 ... */

 return 0;
}
```

Dans cet exemple d'appel, on remarque que la fonction `zero` prend 2 paramètres :
les 2 bornes `0.2` et `10.3`. La fonction `fct` est directement appelée par la fonction `zero`.

TP

1 TP Exercice 1 : petites fonctions mathématiques

1. Programmer l'exercice 3.1 du TD (fonction `poly` + fonction `main`).
   Tester ces fonctions (préconditions, résultats attendus).
2. Programmer l'exercice 3.3 du TD (fonction `myCos`).
   Tester cette fonction et comparer les résultats obtenus avec les résultats de la fonction prédéfine `cos`.

2 TP Exercice 2 : un peu de récursivité

1. En récupérant le code, tester le calcul de la suite de fibonacci récursif et en itératif.
   Tester avec les valeurs suivantes : `0`, `1`, `2`, `4`, `10`, `20`, `30`, `35` et `40`.
   Comparer les temps d'exécution en récursif et en itératif. Avantages et inconvénients de chaque méthode ?

Code à récupérer ici :
`http://lab-sticc.univ-brest.fr/~rodin/FTP/Enseignements/L1/AlgoEtProg`

Il y a une suite au TP... :-)

Pour mémoire en récursif :

```
                                        /* page062_fibonnacciRecursive.c */
#include <stdio.h>   /* Pour printf */
#include <stdlib.h>  /* Pour exit   */

int fibo(int n)
{
 int resultat;

 if (n<0) { printf("Erreur parametre fibo\n");
            exit(1); /* Arret du programme */
 }

 if (n==0 || n==1)
 { resultat = 1;
 }
 else
 { resultat = fibo(n-1) + fibo(n-2);
 }

 return resultat;
}

int main(void)
{
 int n;
 printf("Calcul de la suite de fibonnacci (version recursive)\n");

 printf("Donnez moi une valeur : ");
 scanf("%d",&n);

 printf("fibo(%d)=%d\n",n,fibo(n));

 return 0;
}
```

Pour mémoire en itératif :

```
                                        /* page064_fibonnacciIterative.c */
#include <stdio.h>   /* Pour printf */
#include <stdlib.h>  /* Pour exit   */

int fibo(int n)
{
 int resultat;

 if (n<0)
 {
  printf("Erreur parametre fibo\n");
  exit(1); /* Arret du programme */
 }
```



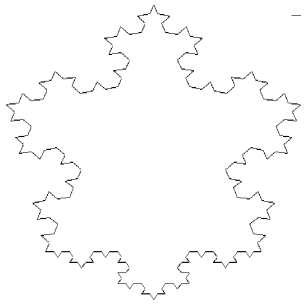
```
 if (n==0 || n==1) { resultat = 1; }
 else
 {
  int indice,un,un_1,un_2;

  un_1   = 1;
  un_2   = 1;
  indice = 2;
  while (indice <= n)
  {
   un   = un_1 + un_2;
   un_2 = un_1;
   un_1 = un;
   indice = indice + 1;
  }
  resultat = un;
 }

 return resultat;
}

int main(void)
{
 int n;
 printf("Calcul de la suite de fibonnacci (version iterative)\n");

 printf("Donnez moi une valeur : ");
 scanf("%d",&n);

 printf("fibo(%d)=%d\n",n,fibo(n));

 return 0;
}
```

2. Dessiner en récursifs : étoile de Von Koch

   Nous désirons réaliser une fonction récursive permettant de dessiner “simplement” l'étoile de Von Koch.

   Pour réaliser cet exercice, récupérer du le projet `VonKoch.zip`, le dcompresser et ouvrir le projet `VonKoch.cbp`.

   Le code à récupérer est sur Moodle ou sur `H:/enseignants/masse/algoprog/`.



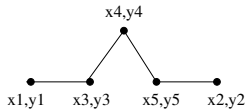
PRINCIPE DE CONSTRUCTION :

- Partir de 3 segments formant un triangle isocèle (les coordonnes de ces segments sont dj calculs dans la fonction principale `graph_main`).
- Pour chaque segment, appeler une **fonction récursive** `segment` permettant de le décomposer en une suite de segments. Sur chacun des segments de cette suite, rappeler la fonction récursivement...
  La profondeur de la récursion (le degré du dessin) doit être passée en paramètre à cette fonction récursive et doit donc permettre d'arrêter la récursion (condition d'arrêt).

  La fonction récursive doit donc avoir le prototype suivant :
  `void segment(int degre, double x1, double y1, double x2, double y2);`
  Cette fonction doit, pour son cas non rcursif , appeler la fonction (dj crite) `vector` suivante :
  `void vector(double x1, double y1, double x2, double y2);`
  Cette fonction, permet de tracer un segment de `(x1,y1)` à `(x2,y2)` (l'chelle dfinie dans la fonction permet normalement de tracer le flocon dans les limites de la fentre).
  Dans le cas rcursif (`degre` > 1) la fonction `segment` dcompose le segment en quatre segments et fait des appels rcursifs sur ces segments.

DÉCOUPAGE DU SEGMENT x1,y1,x2,y2 :

Lorsque le segment x1,y1,x2,y2 doit être décomposé en une suite de segments, appliquer les règles suivantes :



$$\begin{cases} dx &= x2 - x1 \\ dy &= y2 - y1 \end{cases}$$

$$\begin{cases} x3 &= x1 + \frac{dx}{3} \\ y3 &= y1 + \frac{dy}{3} \end{cases}$$

$$\begin{cases} x4 &= x1 + \frac{dx}{2} - dy * \frac{\sqrt{3}}{6} \\ y4 &= y1 + \frac{dy}{2} + dx * \frac{\sqrt{3}}{6} \end{cases}$$

$$\begin{cases} x5 &= x1 + dx * \frac{2}{3} \\ y5 &= y1 + dy * \frac{2}{3} \end{cases}$$

3 TP Exercice 4 :

Eventuellement, programmer les algorithmes restant du TD.