Algorithmes simples (corrigé)

Liste des exercices

Tous les exercices sont à faire. Ils peuvent être réalisés en C (de préférence), en Java (créer une classe publique TP1 et des méthodes statiques pour chaque exercice) ou en Python.
Sont à rendre (par binôme) :
– le programme de l'exercice 3.1
– le programme de l'exercice 4.7
– les programmes des exercices 4.8, 4.9, 4.10 et 4.11

Les exercices sont à envoyer à l'adresse jean-claude.georges@esiee.fr
- objet obligatoire : **IGI-3005 CR TP1 *nom1 nom2***
- listings des fonctions (C, Python) ou méthodes (Java) commentées en pièces jointes (pas de zip)

1 C vs. Python

Voici, sous Linux, la démarche minimale à suivre de l'écriture jusqu'à l'exécution de programmes en C, en Java ou en Python.

	Pour C	Pour Java	Pour Python
1	Ouvrir un terminal	Ouvrir un terminal	Ouvrir un terminal
2	Lancer un éditeur de texte en tâche de fond `> gedit &`	Lancer un éditeur de texte en tâche de fond `> gedit &`	Lancer un éditeur de texte en tâche de fond `> gedit &`

3	Dans l'éditeur, saisir un programme
	`#include <stdio.h>`
	`int main() {`
	`  printf("Bonjour en C");`
	`  return 0;`
	`}`
		Dans l'éditeur, saisir une classe et une méthode
		`public class Essai {`
		`  public static void main(`
		`  String [] args) {`
		`  System.out.println(`
		`  "Bonjour en Java");`
		`  }`
		`}`
			Dans l'éditeur, saisir un programme
			`print("Bonjour en Python")`

4	Sauvegarder le programme : `essai.c`	Sauvegarder la classe : `Essai.java` *(attention à la majuscule)*	Sauvegarder le programme : `essai.py`
5	Lancer la compilation `> cc essai.c - o essai` Si erreur, retourner à l'étape 3	Lancer la compilation `> javac Essai.java` Si erreur, retourner à l'étape 3	*Pas de compilation*
6	Lancer le programme : `> ./essai`	Lancer le programme : `> java Essai`	Lancer le programme : `acme1:~> python3 -m essai`
7	Admirer le résultat : `Bonjour en C` Si erreur, retourner à l'étape 3	Admirer le résultat : `Bonjour en Java` Si erreur, retourner à l'étape 3	Admirer le résultat : `Bonjour en Python` Si erreur, retourner à l'étape 3

Exercice 1. Comparer C et Python (*)

Trois programmes écrits en C, en Java et en Python permettent de calculer le nombre de solutions au problème des huit reines : huit_reines.c, HuitReines.java et huit_reines.py (voir wikipedia sur les huit reines)

Téléchargez ces programmes et suivez la démarche ci-dessus pour exécuter les trois programmes.

Lequel est le plus rapide ? De beaucoup ?

Modifiez les programmes sources pour qu'ils affichent les résultats de 1 à 10 au lieu de 1 à 12. Quelle modification se fait le plus rapidement ?

2 Les bases de l'écriture de programmes

2.1 Conversion kilomètres-miles (*)

Exercice 2. Écrire une fonction qui convertit les kilomètres en miles (1 mile = 1,609 km).

Prototype C : `double km_vers_mile(double km);`

Prototype Java : `public static double km_vers_mile(double km);`

Prototype Python : `def km_vers_mile(km) :`

Corrigé

****

Attention au petit piège. 1 mile = 1,609 km $\Rightarrow$ il faut **diviser** le nombre de kilomètres par $1,609$ pour obtenir le nombre de miles.

```
double km_vers_mile(double km) {
    return km/1.609;
}
```

****

2.2 Conversion Fahrenheit - Centigrade (*)

Exercice 3. Écrire une fonction qui convertit les degrés Fahrenheit en degrés centigrades.

Formule : $\theta_{\mathrm{C}} = \frac{5}{9}(\theta_{\mathrm{F}} - 32)$

Prototype C : `double F_vers_C(double f);`

Prototype Java : `public static double F_vers_C(double f);`

Prototype Python : `def F_vers_C(f) :`

Corrigé

****

Le seul écueil à éviter (en C) est de ne pas coder la division $5 \div 9$ par `5/9`, qui vaut `0` en langage C.

```
double F_vers_C(double f) {
    return 5.0/9.0 * (f-32.0);
}
```

****

2.3 Volume d'une sphère (*)

Exercice 4. Écrire une fonction qui calcule le volume d'une sphère.

Étant donné le rayon R, le volume d'une sphère est donné par $V = \frac{4}{3}\pi R^3$

Prototype C : `double volume_sphere(double rayon);`

Prototype Java : `public static double volume_sphere(double rayon);`

Prototype Python : `def volume_sphere(rayon):`

Corrigé

****

Il ne faut pas coder la division $4 \div 3$ par `4/3`, qui vaut `1` en langage C.
D'autre part, la valeur de π est présente dans `math.h` :

```
#define M_PI 3.14159265358979323846
```

Il vaut mieux inclure `math.h` (sauf si l'on connaît par cœur la valeur de π, et que l'on est sûr de ne pas faire de faute de frappe).

```
#include <math.h>
double volume_sphere(double rayon) {
    return 4.0/3.0 * M_PI * rayon*rayon*rayon;
   /* ou 4.0/3.0 * M_PI * pow(rayon,3);
      en compilant avec l'option -lm (lib. math.) */
}
```

****

3 Conditionnelles

3.1 Salaire (*)

Exercice 5. Écrire une fonction ayant en paramètres le nombre d'heures effectuées par un salarié dans la semaine et son salaire horaire, qui retourne sa paye hebdomadaire.

On prendra en compte les heures supplémentaires (au-delà de 35 heures) payées à 150%.

Prototype C : `double salaire_hebdomadaire(int nb_heures, double salaire_horaire);`

Prototype Java : `public static double salaire_hebdomadaire(int nb_heures, double salaire_horaire);`

Prototype Python : `def salaire_hebdomadaire(nb_heures, salaire_horaire):`

Corrigé

****

Ici, éviter à tout prix d'écrire *en dur* 35 et 150 dans le code (*magic numbers*). Utiliser des `##define` pour pouvoir plus facilement modifier le programme (les lois et les taux peuvent changer).

```
#define DUREE_LEGALE 35
#define TAUX_MAJ_HEUR_SUP 150 /* 150 % */
double salaire_hebdo(int nb_heures, double salaire_horaire) {
    double res;
    if (nb_heures < DUREE_LEGALE) {
        res = nb_heures*salaire_horaire;
    }
    else {
        res =   (DUREE_LEGALE
                    +
                        (nb_heures - DUREE_LEGALE)
                        *
                        (TAUX_MAJ_HEUR_SUP / 100.0)
                )
                *
                salaire_horaire;
    }
    return res;
}
```

****

3.2 Notes (*)

Un professeur note les résultats d'un test portant sur 50 questions en utilisant la table suivante :

bonnes réponses	0-10	11-20	21-30	31-40	41-50
note	E	D	C	B	A

Exercice 6. Écrire une fonction qui retourne la note, étant donné un nombre bonnes réponses.

Prototype C : `char note (int bonnes_reponses);`

Prototype Java : `public static char note (int bonnes_reponses);`

Prototype Python : `def note (bonnes_reponses):`

Corrigé

****

Un programme satisfaisant, quoique contenant des *magic numbers* (mais il est tellement simple qu'une modification du barème pourrait être facilement répercutée) pourrait être le suivant :

```
char note1 (int bonnes_reponses) {
    if (bonnes_reponses<0 || bonnes_reponses>50)
        return '#';
    else if (bonnes_reponses <= 10)
        return 'E';
    else if (bonnes_reponses <= 20)
        return 'D';
    else if (bonnes_reponses <= 30)
        return 'C';
    else if (bonnes_reponses <= 40)
        return 'B';
    else if (bonnes_reponses <= 50)
        return 'A';
}
```

Le programme suivant, horrible et à ne jamais écrire, répond aux spécifications. Il est difficilement lisible, une modification du barème entraînerait une maintenance coûteuse, il utilise un nombre en octal (`0105` $= 69$), pour représenter un caractère ('E'), et un nombre en hexadécimal (`0xa` $= 10$). Il soustrait donc à 'E' le quotient de la division entière par 10 du nombre de bonnes réponses moins un.

```
/*********************************************
    NE JAMAIS ECRIRE CE GENRE DE PROGRAMME
*********************************************/
char note2(int bonnes_reponses) {
    return 0105 - --bonnes_reponses/0xa;
}
```

Pour ne pas avoir à utiliser de *magic numbers*, et pouvoir modifier facilement le barème, il est nécessaire de disposer de structures et de tableaux. Le programme suivant conviendrait :

```
char note3(int bonnes_reponses) {
    static struct {
        int borne_inf, borne_sup;
        char note;
    } bareme[]={ {0, 10, 'E'},
                 {11, 20, 'D'},
                 {21, 30, 'C'},
                 {31, 40, 'B'},
                 {41, 50, 'A'}};
    int i;
    for (i=0 ; i < sizeof(bareme)/sizeof(bareme[0]) ; ++i) {
        if (bareme[i].borne_inf <= bonnes_reponses
                &&
                bonnes_reponses <= bareme[i].borne_sup)
            return bareme[i].note;
    }
    return'#';
}
```

****

3.3 Calendrier (**)

Parmi les traitements les plus souvent utilisés en informatique, on touve sans doute ceux agissant sur des dates (affichage et calcul).

Le calendrier en usage en France est le calendrier dit *grégorien* basé sur les principes suivants :

- l'année est divisée en 12 mois numérotés de 1 à 12 ;
- les mois 1, 3, 5, 7, 8, 10 et 12 comptent 31 jours ;
- les mois 4, 6, 9 et 11 ont comptent 30 jours ;
- le mois 2 compte 29 jours si l'année est *bissextile*, 28 sinon
- une année est bissextile si
  - elle ne se termine pas par 00 et est divisible par 4,
  - elle se termine par 00 et son quotient par 100 est divisible par 4.

Exercice 7. Écrire une fonction ayant en paramètre un entier représentant une année et retournant `1` (`true`, `True`) si l'année est bissextile et `0` (`false`, `False`)sinon.

Prototype C : `int bissextile(int annee);`

Prototype Java : `public static boolean bissextile(int annee);`

Prototype Python : `def bissextile(annee):`

Corrigé

****

```c
int bissextile(int annee) {
    return annee%4 == 0 && ((annee%100 != 0) || (annee%400 == 0));
}
```

****

Exercice 8. Écrire une fonction ayant en paramètre deux entiers représentant un mois et une année et retournant le nombre de jours du mois de cette année.

Prototype C : `int nombre_jours_mois(int mois, int annee);`

Prototype Java : `public static int nombre_jours_mois(int mois, int annee);`

Prototype Python : `def nombre_jours_mois(mois, annee):`

Corrigé

****

```c
int nombre_jours_mois(int mois, int annee) {
    switch (mois) {
        case 1 : case 3 : case 5 :
        case 7 : case 8 : case 10 :
        case 12 :
            return 31;
        case 4 : case 6 : case 9 :
        case 11 :
            return 30;
        case 2 :
            if (bissextile(annee))
                return 29;
            else
                return 28;
        default :
            return 0;
    }
}
```

****

Exercice 9. Écrire une fonction ayant en paramètre trois entiers représentant un jour, un mois et une année et retournant 1 s'ils représentent une date valide et 0 sinon.

Le calendrier grégorien ayant été appliqué en France en décembre 1582, on considérera les dates antérieures à 1583 comme non invalides.

Prototype C : `int datevalide(int j, int m, int a);`

Prototype Java : `public static int datevalide(int j, int m, int a);`

Prototype Python : `def datevalide(j, m, a):`

Corrigé

****

```
int datevalide(int j, int m, int a) {
    return (a >= 1583
                && 1 <= m && m <= 12
                && 1 <= j && j <= nombre_jours_mois(m, a));
}
```

****

4 Boucles et récursivité

4.1 PGCD (*)

Exercice 10. Écrire une fonction qui retourne le plus grand commun diviseur (*pgcd*) de deux nombres entiers positifs.

L'algorithme d'Euclide est basé sur le principe suivant :

$$\text{pgcd}(a,b) = \begin{cases} a & \text{si } b = 0 \\ \text{pgcd}(b, a \bmod b) & \text{sinon} \end{cases}$$

Prototype C : `int pgcd(int a, int b);`

Prototype Java : `public static int pgcd(int a, int b);`

Prototype Python : `def pgcd(a, b):`

Corrigé

****

Deux solutions, l'une itérative et l'autre récursive, sont possibles :

```
int pgcd_iter(int a, int b) {
    int c;
    while (b!=0) {
        c=a %b;
        a=b;
        b=c;
    }
    return a;
}
```

ou bien

```
int pgcd_rec(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
```

```
        return pgcd_rec(b, a%b);
    }
}
```

****

4.2 Factorielle (*)

Exercice 11. Écrire une fonction qui retourne la factorielle d'un entier.

On pourra s'inspirer des deux premières versions de `somme_nombres` vues en préambule.

$$\text{n!} = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 & \text{sinon} \end{cases}$$

Que se passe-t-il en cas de dépassement de capacité (C et Java) ?

Prototype C : `int factorielle(int n);`

Prototype Java : `public static int factorielle(int n);`

Prototype Python : `def factorielle(n):`

Corrigé

****

Une première solution pourrait être basée sur le principe suivant :

- on initialise une variable (accumulateur) à 1
- on exécute une boucle dans laquelle l'accumulateur est multiplié par une variable prenant les valeurs de 1 à n, le résultat étant stocké dans l'accumulateur
- en fin de boucle, le résultat est dans l'accumulateur, ce qui donne :

```
int factorielle_iter(int n) {
    int i;
    int f = 1;
    for (i=1 ; i<=n ; ++i) {
        f *= i;
    }
    return f;
}
```

Une deuxième solution, sur le même principe, mais utilisant le paramètre de la fonction comme compteur, est la suivante (cette version est déconseillée car elle est nettement plus difficile à comprendre)

```
int factorielle_iter2(int n) {
    int f = 1;
    while (n) {
        f *= n--;
    }
    return f;
}
```

Enfin, une solution récursive peut être envisagée, basée sur la définition mathématique.

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Elle s'écrit en traduisant quasiment mot à mot la définition mathématique :

```
int factorielle_rec (int n) {
    if (n <= 0) {
        return 1;
```

```
    }
    else {
        return n * factorielle_rec(n-1);
    }
}
```

Ces trois versions retournent 1 si l'argument d'appel est négatif. Mais le plus grand problème est qu'elles retournent des résultats faux pour des valeurs d'arguments trop grandes :

13! est évalué à 1 932 053 504, alors que sa valeur exacte est 6 227 020 800. Ce qui se passe, c'est que le calcul de 13! dépasse la valeur de l'entier maximum autorisé (avec mon compilateur $2^{31} - 1 = 2\,147\,483\,647$), valeur que l'on peut trouver selon sa plate-forme à la ligne de `limits.h` :

```
#define INT_MAX     2147483647
```

Cette erreur ne peut pas être corrigée, sauf à changer la spécification de la fonction (un retour `double`) avec des valeurs approchées à partir de 22! et un dépassement de capacité à partir de 171! :

```
double factorielle_dbl(int n) {
    int i;
    double f = 1.0;
    for (i=1 ; i<=n ; ++i) {
        f *= i;
    }
    return f;
}
```

ou en utilisant ou en écrivant une bibliothèque C de calcul sur les grands nombres.

****

4.3 Plus petit diviseur premier (**)

Exercice 12. Écrire une fonction qui retourne le plus petit diviseur premier d'un nombre.

Prototype C : `int plus_petit_diviseur_premier (int n);`

Prototype Java : `public static int plus_petit_diviseur_premier (int n);`

Prototype Python : `def plus_petit_diviseur_premier (n):`

Principe : tester le reste de la division de n par tous les nombres à partir de 2.

Corrigé

****

Une première tentative peut être écrite en suivant le principe d'essayer de diviser le nombre argument par les diviseurs potentiels. On commence par 2 et tant que la division n'est pas exacte, on passe au diviseur potentiel suivant (on est sûr d'en trouver au moins un, le nombre argument), ce qui donne :

```
int plus_petit_diviseur_premier (int n) {
    int d;
    for (d=2 ; n%d != 0 ; ++d) {
    /* on ne fait rien */
    }
    return d;
}
```

Une curiosité de ce programme est que la boucle ne fait "rien", les instructions de la boucle (initialisation, condition, incrémentation) font tout le travail.

Il apparaît toutefois que l'on fait trop de divisions. En effet, si l'on n'a pas trouvé de diviseurs premiers inférieurs ou égaux à $\sqrt{n}$, alors le nombre n est premier (s'il y avait un diviseur p supérieur à $\sqrt{n}$, le quotient de $n \div p$ serait entier et inférieur à $\sqrt{n}$ et on aurait dû le trouver).

Une seconde version utilise ce principe :

```c
int plus_petit_diviseur_premier2 (int n) {
    int d;
    for (d=2 ; (d*d <= n) && (n%d != 0) ; ++d) {
    /* on ne fait toujours rien */
    }
    if (n%d == 0) {
        return d;
    }
    else {
        return n;
    }
}
```

Enfin, une troisième version met à par le cas 2, pour éviter les divisions par les nombres pairs.

```c
int plus_petit_diviseur_premier3 (int n) {
    int d;
    if (n%2 == 0) {
        return 2;
    }

    for (d=3 ; (d*d <= n) && (n%d != 0) ; d+=2) {
    }
    if (n%d == 0) {
        return d;
    }
    else {
        return n;
    }
}
```

****

4.4 Racine carrée entière (**)

Exercice 13. Écrire une fonction qui calcule la racine carrée entière d'un nombre entier positif.

Principe : on opérera par soustractions successives des nombres impairs. En effet, si p est l'entier solution de la double inéquation

$$\sum_{i=1}^{p}(2i-1) \leq n < \sum_{i=1}^{p+1}(2i-1)$$

alors $p \leq \sqrt{n} < p+1$ et donc p est la racine carrée entière de n.

Exemple : racine de 43

$43-1=42,\ 42-3=39,\ 39-5=34,\ 34-7=27,\ 27-9=18,\ 18-11=7$

La prochaine soustraction, $7-13$ donnerait un résultat négatif, on n'a pu faire que 6 soustractions en tout, donc la racine entière de 43 est 6.

Prototype C : `int racine_entiere(int n);`

Prototype Java : `public static int racine_entiere(int n);`

Prototype Python : `def racine_entiere(n):`

Corrigé

****

Une fois le principe appréhendé, le programme n'est pas très difficile à écrire.

```c
int racine_entiere(int n) {
```

```
    int impair = 1, nbsoustractions = 0;
    while (impair <= n) { /* tant que soustraction possible */
        n -= impair; /* on la fait */
        ++nbsoustractions; /* on la compte */
        impair += 2; /* on passe à l'impair suivant */
    }
    return nbsoustractions;
}
```

****

4.5 Racine carrée (**)

Exercice 14. Écrire une fonction qui calcule la racine carrée d'un nombre réel positif.

L'algorithme de Newton est basé sur la convergence de la suite suivante :

Principe : la suite définie par

$$\begin{cases} u_0 = 1 \\ u_{n+1} = \dfrac{u_n + \dfrac{a}{u_n}}{2} \end{cases}$$

converge vers $\sqrt{a}$.

Prototype C : `double racine_carre(double x);`

Prototype Java : `public static double racine_carre(double x);`

Prototype Python : `def racine_carre(x):`

Corrigé

****

La programmation d'une boucle calculant les termes de la suite de Newton semble évidente (en faisant bien attention à ne pas écrire `1/2` qui serait évalué à `0`, mais `0.5`) :

```
while (...) {
    nouveau_terme=0.5*(ancien_terme+x/ancien_terme);
}
```

La difficulté est de savoir quand arrêter la boucle. Utiliser une boucle `for` avec une borne maximum constante n'est pas une bonne idée. Est-on sûr qu'à 100, 1 000, etc., la suite aura convergé ? ou au contraire sera-t-elle encore loin de la convergence ?

Une meilleure solution consiste à arrêter la boucle lorsque les deux derniers termes de la suite seront suffisamment proches.

Attention : ne jamais faire de de tests d'égalité `==` ou `!=` sur les `double`, la suite pouvant osciller entre deux valeurs très proches.

Il faudra donc programmer un test de proximité de deux nombres selon le principe :

- si les deux nombres sont très proches de 0, ils seront considérés comme égaux. Les `double` étant limités à des *petits nombres* de l'ordre de 10^{-300}, on pourra considérer par exemple que deux nombres inférieurs en valeur absolue 10^{-200} sont nuls et donc égaux (mais évidemment, cela dépend du contexte)
- si l'un des deux nombres est très différent de zéro, on considérera comme lui étant égal tout nombre avec lequel l'écart **relatif** sera inférieur en valeur absolue à une certaine valeur. Les `double` étant codés à environ 15 chiffres significatifs, on pourra considérer que deux nombres sont égaux si leur écart relatif est inférieur en valeur absolue à 10^{-10} (mais là aussi, cela dépend du contexte).

Ce qui donne le programme suivant :

```c
/* ... */
#include <math.h> /* pour fabs */
/* pour l'égalité de deux doubles
* - redéfinir epsilonZero à la valeur au dessous
* de laquelle on considère un nombre comme nul
* - redéfinir epsilonPrecision selon le nombre de
* chiffres significatifs que l'on veur avoir pour l'égalitét
*/
#define epsilonZero (1e-200)
#define epsilonPrecision (1e-10)

int egale(double a, double b) {
    if (fabs(a) < epsilonZero) {
        return (fabs(b) < epsilonZero);
    }
    else {
        return (fabs(a-b) < epsilonPrecision*fabs(a));
    }
}

double racine_carree(double a) {
    double u2 = 1.0, u1;
    do {
        u1 = u2;
        u2 = (u1+a/u1) * 0.5;
    } while (! egale(u1, u2);
    return u1;
}
```

****

4.6 Suite bizarre (**)

Exercice 15. Écrire une fonction qui calcule le n^e terme de la suite suivante :

$$\left| \begin{array}{l} u_0 = \dfrac{1}{3} \\ u_{n+1} = 4u_n - 1 \end{array} \right.$$

Tester pour $n = 100$. Quel est le résultat théorique ? Expliquer l'écart.

Prototype C : `double suite_bizarre(int n);`

Prototype Java : `public static double suite_bizarre(int n);`

Prototype Python : `def suite_bizarre(n):`

Corrigé

****

Le programme s'écrit facilement :

```c
double suite_bizarre(int n) {
    int i; double u;
    u = 1.0 / 3.0;
    for (i=1 ; i<=n ;++i) {
        u = 4.0*u/3.0 - 1.0;
    }
    return u;
}
```

Or, en le testant, on constate que

u_{100} est évalué à $-8.314\,619\,760\,551\,515 \cdot 10^{12}$,

ce qui conduirait à penser que la limite de la suite serait $-\infty$. Mais un simple calcul montre que la suite est constante et reste égale à $1/3$ et donc que le résultat est faux.

L'explication vient du fait que les calculs se font à une précision limitée et que les erreurs de calcul sur les derniers chiffres significatifs s'accumulent. Une visualisation en base 10 le montre assez bien :

Supposons que les résultats des opérations soient arrondis à trois chiffres significatifs.

$$\begin{array}{lcrcr} u_0 & & & = & 0,333 \\ u_1 & = & 4 \times 0,333 - 1 & = & 0,332 \\ u_2 & = & 4 \times 0,332 - 1 & = & 0,328 \\ u_3 & = & 4 \times 0,328 - 1 & = & 0,312 \\ u_4 & = & 4 \times 0,312 - 1 & = & 0,248 \\ u_5 & = & 4 \times 0,248 - 1 & = & -0,008 \\ u_6 & = & 4 \times -0,008 - 1 & = & -1,03 \end{array}$$

et on constate que l'on se met en route vers $-\infty$. Il se passe exactement la même chose en machine.

En informatique théorique (ou en mathématique), on montre que la suite $u_{n+1} = f(u_n)$ converge si la dérivée de f prend des valeurs inférieures à 1 (en valeur absolue) autour de la limite, ce qui n'est pas le cas ici ($f(x) = 4x - 1$ et $f'(x) = 4$).

Moralité : même un problème simple ne doit pas se programmer sans précaution.

****

4.7 Suite de Fibonacci (**)

Exercice 16. Écrire une fonction qui retourne le n^{e} terme d'une suite de Fibonacci initialisée par a et b.

$$\left| \begin{array}{lcl} u_0 & = & a \\ u_1 & = & b \\ u_n & = & u_{n-1} + u_{n-2} \quad \text{pour } n \geq 2 \end{array} \right.$$

Prototype C : `int fibonacci(int n, int a, int b);`

Prototype Java : `public static int fibonacci(int n, int a, int b);`

Prototype Python : `def fibonacci(n, a=0, b=1):`

Corrigé

****

Deux versions peuvent s'écrire, l'une itérative, l'autre récursive. La plus facile à écrire est la récursive, qui traduit en C la définition mathématique de la suite quasiment mot à mot :

```
int fibonacci_rec(int n, int a, int b) {
    switch(n) {
        case 0 :
            return a;
        case 1 :
            return b;
        default :
            return fibonacci_rec(n-1, a, b) + fibonacci_rec(n-2, a, b);
    }
}
```

Malheureusement, si elle est facile à écrire, elle est d'une lenteur désespérante à l'exécution (essayer `fibonacci_rec(50)` , même sur un ordinateur récent).

L'explication tient au fait que pour calculer u_{50}, on calcule déjà u_{49} et u_{48}, qui eux mêmes nécessitent le calcul de u_{48} et u_{47} pour l'un et u_{47} et u_{46} pour l'autre, etc. Le temps de calcul devient assez vite prohibitif.

Essayons d'écrire une fonction itérative.

La version itérative est plus difficile à écrire : on traite bien entendu les cas 0 et 1 à part, et dans le cas général, une boucle dans laquelle :

- on calcule le nouveau terme en fonction du dernier et de l'avant-dernier ;
- on modifie les valeurs du dernier terme et de l'avant-dernier terme pour préparer le tour de boucle suivant.

Il faut être très attentif à l'ordre des instructions, de manière à ne pas perdre de valeurs utiles à la suite du calcul.

```
int fibonacci_iter(int n, int a, int b) {
    int i;
    int avantDernier, dernier, suivant;
    switch(n) {
        case 0 :
            return a;
        case 1 :
            return b;
        default :
            avantDernier = a;
            dernier = b;
            for (i=2;i<=n;++i) {
                suivant = avantDernier + dernier;
                avantDernier = dernier;
                dernier = suivant;
            }
            return suivant;
    }
}
```

Cette version peut être réécrite en une version récursive acceptable :

```
int fibonacci_rec2(int n, int a, int b) {
    int i;
    switch(n) {
        case 0 :
            return a;
        case 1 :
            return b;
        default :
            return fibonacci_rec2(n-1, b, a+b)
    }
}
```

****

4.8 Puissance entière itérative (**)

Exercice 17. Écrire une fonction qui calcule a^n avec a réel et n entier positif. Utiliser un algorithme *itératif*.

Prototype C : `double puissance(double a, int n);`

Prototype Java : `public static double puissance(double a, int n);`

Prototype Python : `def puissance(a, n):`

Note : on s'interdira bien évidemment d'utiliser la fonction `pow` de la librairie mathématique.

Corrigé

****

Une simple boucle suffit (schéma d'accumulation).

```
double puissance(double x, int n) {
```

```
    double res;
    int i;
    res = 1.0;
    for (i=1 ; i<=n ; ++i) {
        res *= x;
    }
    return res;
}
```

La complexité est en $O(n)$, aussi éléver à la puissance $1\,000\,000\,000$, par exemple, est assez long.
****

4.9 Puissance entière récursive (**)

Exercice 18. Écrire une fonction qui calcule a^n avec a réel et n entier positif.Utiliser un algorithme *récursif*.

Le principe en est le suivant :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ (a^{n/2})^2 & \text{si } n \text{ est pair} \\ a \cdot a^{n-1} & \text{si } n \text{ est impair} \end{cases}$$

Comparer les temps d'exécution avec la fonction précédente pour $n = 100, 1000, 10000$ (pour éviter les dépassements de capacité, prendre une valeur de test pour a proche de 1)

Prototype C : `double puissance_rapide(double a, int n);`

Prototype Java : `public static double puissance_rapide(double a, int n);`

Prototype Python : `def puissance_rapide(a, n):`

Note : on s'interdira bien évidemment d'utiliser la fonction `pow` de la librairie mathématique.

Corrigé

****

Sur le principe de l'énoncé, le programme est un peu plus difficile à écrire :

- cas terminal : si n est nul : `return 1;`
- cas général
  - si n impair : on écrit facilement : `return a * puissance_rapide(a,n-1);`
  - si n pair : une première fausse bonne idée serait de *surutiliser* la récursivité en écrivant par exemple :
    `return puissance_rapide(puissance_rapide(a, n/2), 2)`
    On entrerait ainsi dans une récursivité sans fin lorsque n atteindrait la valeur 2 (puissance 2 appellerait puissance 2 qui appellerait puissance 2 etc.)
    Une deuxième fausse bonne idée serait d'écrire une fonction juste, du type
    `return puissance_rapide(a, n/2) * puissance_rapide(a, n/2)`
    mais qui ne tirerait pas parti des appels déjà effectués. En effet, a^{50} par exemple appellerait deux fois a^{25}.

Finalement, il vaut mieux écrire quelque chose du style :

```
double puissance_rapide(double x, int n) {
    double tmp;
    if (n == 0) {
        return 1;
    }
    else {
```

```
        if (n%2 == 0) {
            tmp=puissance_rapide(x, n/2);
                return (tmp * tmp);
        }
        else {
            return (x * puissance_rapide(x, n-1));
        }
    }
}
```

La complexité est en $O(\log(n))$: éléver à la puisance $1\,000\,000\,000$, par exemple, est très rapide.

Cette version récursive peut être *transformée* en une verion itérative de même complexité, mais plus difficile à écrire :

```
double puiss(double x, int n) {
    double t = x;
    int p = n;
    double res = 1.0 ; /* x**n = res * t^p */
    while (p > 0) {
        if (p%2==0) { /*  res * t**p = res * (t*t)**(p/2)*/
            t *= t;
            p /= 2; /* x**n = res * t**p */
        }
        else { /* res * t**p = (res * t) * t**(p-1) */
            res *= t;
            p -= 1; /* x**n = res * t**p */
        }
    }
    /* x**n = res * t**p et p==0 */
    return res;
}
```

Note : pour tester cette fonctions pour des grandes valeurs d'exposants (et pour vérifier les résultats), on peut se servir du fait que

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = e$$

****

4.10 Développements limités (**)

Exercice 19. Écrire les fonctions qui calculent les développements limités à l'ordre *n* de sin *x*, cos *x*, e^x^ et arctan *x*.

Rappel :

$$\left|\begin{array}{lcl}
\sin x & = & x - \dfrac{x^3}{3!} + \dfrac{x^5}{5!} - \dfrac{x^7}{7!} + \dots \\
\cos x & = & 1 - \dfrac{x^2}{2!} + \dfrac{x^4}{4!} - \dfrac{x^6}{6!} + \dots \\
\mathrm{e}^x & = & 1 + x + \dfrac{x^2}{2!} + \dfrac{x^3}{3!} + \dfrac{x^4}{4!} + \dots \\
\arctan x & = & x - \dfrac{x^3}{3} + \dfrac{x^5}{5} - \dfrac{x^7}{7} + \dots
\end{array}\right.$$

Prototype C : `double dev_lim_sin(double x, int n);`

Prototype C : `double dev_lim_cos(double x, int n);`

Prototype C : `double dev_lim_exp(double x, int n);`

Prototype C : `double dev_lim_atn(double x, int n);`

Prototype Java : `public static double dev_lim_sin(double x, int n);`

Prototype Java : `public static double dev_lim_cos(double x, int n);`

Prototype Java : `public static double dev_lim_cos(double x, int n);`

Prototype Java : `public static double dev_lim_atn(double x, int n);`

Prototype Python : `def dev_lim_sin(x, n):`

Prototype Python : `def dev_lim_cos(x, n):`

Prototype Python : `def dev_lim_exp(x, n):`

Prototype Python : `def dev_lim_atn(x, n):`

Corrigé

****

Une première version (on discutera uniquement sur `dev_lim_sin`) consiste à utiliser un schéma d'accumulation recopiant quasiment mot à mot la formule mathématique :

```c
/* utilitaire calcul de factorielle en double,
   evitant le dépassement de capacité entier
*/
double fact(int n) {
    double res = 1.0;
    for (res = 1.0 ; n > 0 ; n--) {
        res *= n;
    }
    return res;
}

/* version du developpement limité de sinus,
   utilisant la fonction pow de la bibliothèque mathématique
   Nécessite l'inclusion de <math.h> et l'édition de liens
   avec la librairie mathématique libm (gcc -lm ...)
*/
double dev_lim_sin1(double x, int n) {
    double res = 0.0; int i;
    for (i=1 ; i <= n ;i += 2) {
        res += pow(-1.0, i/2) * pow(x, i) / fact(i);
    }
    return res;
}
```

Il apparaît toutefois que le caclul de $(-1)^{\mathrm{Ent}(i/2)}$ utilisant la fonction `pow` est gaspilleur de capacité de calcul (cette puissance vaut alternativement -1 et $+1$, et une variable entière représentant le signe sera plus économique). D'autre part, les calculs répétés de x^i et $i!$ ne réutilisent pas les calculs précédents (x^{15} suit le calcul de x^{13}, sans utiliser son résultat, alors qu'une simple multiplication par x^2 suffirait ; il en est de même pour $15!$)

Deux variables mémorisant les puissances et factorielles seront là aussi plus efficaces. D'où une deuxième version, plus rapide :

```c
double dev_lim_sin2(double x, int n) {
    double res=0.0,
           x2 = x*x, /* évite de recalculer x*x à chaque tour */
           mem_puiss, /* dernière puissance calculée */
           mem_fact; /*dernière factorielle calculée */
    int i,
        signe = 1; /* vaudra alternativement -1 et +1 */
```

```
    mem_puiss = x;
    mem_fact = 1.0;
    x2 = x*x;
    for (i = 1 ; i <= n ; i += 2) {
        res += signe * mem_puiss / mem_fact;
        signe = -signe;
        mem_puiss *= x2;
        mem_fact *= (i+1) * (i+2);
    }
    return res;
}
```

On pourrait encore tenter d'améliorer l'efficacité en utilisant par exemple le schéma de calcul de Horner (1786 – 1837) :

$$a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0 = (\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$$

Cette troisième version est laissée à titre d'exercice. Toutefois, ce que l'on gagnera en efficacité se perdra en lisibilité. N'utilisez ce genre de programmation que si le critère d'efficacité est primordial.

Voici les codes des trois autres développements limités demandés :

```
double dev_lim_cos(double x, int n) {
    double res = 0.0,
           x2 = x*x, /* evite de recalculer x*x a chaque tour */
           mem_puiss,
           mem_fact;
    int i, signe = 1;

    mem_puiss = 1.0;
    mem_fact = 1.0;
    x2 = x*x;
    for (i = 0 ; i <= n ; i += 2) {
        res += signe * mem_puiss / mem_fact;
        signe = -signe;
        mem_puiss *= x2;
        mem_fact *= (i+1) * (i+2);
    }
    return res;
}
```

```
double dev_lim_exp(double x, int n) {
    double res = 0.0,
           mem_puiss,
           mem_fact;
    int i;
    mem_puiss = 1.0;
    mem_fact = 1.0;
    for (i = 0 ; i <= n ; ++i) {
        res += mem_puiss/mem_fact;
        mem_puiss *= x;
        mem_fact *= i+1;
    }
    return res;
}
```

```
double dev_lim_atn(double x, int n) {
    double res = 0.0,
           x2 = x*x, /* evite de recalculer x*x a chaque tour */
           mem_puiss;
    int i, signe = 1;
    mem_puiss = x;
    x2 = x*x;
    for (i = 1 ; i <= n ; i += 2) {
```

```
        res += signe * mem_puiss / i;
        signe = -signe;
        mem_puiss *= x2;
    }
    return res;
}
```

Il est à noter que les logiciels de calcul (ou les calculatrices) n'utilisent pas les développements limités, mais l'algorithme CORDIC **CO**ordinate **R**otation **DI**gital **C**omputer (voir http://www.jacques-laporte.org/LeSecretDesAlgorithmes.htm)

****

4.11 Calcul de π (***)

Exercice 20. Trois méthodes pour calculer une valeur approchée de π.

• Écrire une fonction qui calcule une valeur approchée de π par le calcul du développement limité à l'ordre n de $\arctan 1 (= \frac{\pi}{4})$, multiplié par 4. Jusqu'à quelle valeur de n faut-il pousser le calcul pour avoir trois chiffres significatifs ?

• Écrire une fonction qui calcule une valeur approchée de π par le calcul des développements limités à l'ordre n correspondant à la formule de Machin $\left(\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}\right)$. Jusqu'à quelle valeur de n faut-il pousser le calcul pour avoir trois chiffres significatifs ?

• Dans les années 1970, Salamin trouve une méthode plus rapide basée sur la théorie des intégrales elliptiques. Dans cet algorithme, la n^{e} valeur approchée de π se calcule par :

$$\pi_n = \frac{4a_n^2}{1 - 2\sum_{i=1}^{n} 2^i \left(a_i^2 - b_i^2\right)} \text{ avec } \left| \begin{array}{ll} a_0 = 1 & b_0 = \dfrac{1}{\sqrt{2}} \\ a_n = \dfrac{a_{n-1} + b_{n-1}}{2} & b_n = \sqrt{a_{n-1} b_{n-1}} \end{array} \right.$$

Écrire une fonction qui calcule une valeur approchée de π par le la méthode de Salamin. Jusqu'à quelle valeur de n faut-il pousser le calcul pour avoir dix chiffres significatifs ?

Note : le record actuel (17 octobre 2011) de décimales de π est de 10 billions (10^{13} décimales) http://www.numberworld.org/misc_runs/pi-10t/details.html

Corrigé

****

```
double calpi1(int n) {
    return 4 * dev_lim_atn(1.0, n);
}
```

C'est pour $n = 637$ que π vaut environ $3,144\,727$ et que l'écart relatif devient inférieur à 10^-3 (environ $9,978 \cdot 10^{-4}$)

```
/* formule de Machin
Pi = 4 (4 arctan(1/5) - Arctan(1/239))
*/
double calpi2(int n) {
    return 4 *
                (4 * dev_lim_atn(1.0/5, n)
                    - dev_lim_atn(1.0/239, n)));
}
```

Dès $n = 3$ (trois), π vaut environ $3,140\,597$ et l'écart relatif devient inférieur à 10^-3 (environ $-3.169 \cdot 10^{-4}$)

Moralité : une petite étude mathématique peut faire gagner beaucoup de temps...

Enfin, pour la méthode de Salamin

```
/* formule de Salamin
a0=1
b0=1\rac(2)
a_n=(a_(n-1)+b_(n-1))/2
b_n=rac(a_(n-1)*b_(n-1))

Pi_n=(4(a_n)^2)/(1-2sigma_(i=1..n)(2^n*(a^2-b^2)))
*/

double calpi3(int n) {
    double a = 1.0,
           b = 1/sqrt(2.0),
           s = 0.0,
           p2 = 1.0,
           t;
    int i;
    for (i = 1 ; i <= n; ++i) {
        p2 *= 2;
        t = a;
        a = (a+b) / 2;
        b = sqrt(t*b);
        s += p2 * (a*a-b*b);
    }
    return 4 * a * a / (1-2*s);
}
```

Dès $n = 3$ (trois), l'écart à π est d'environ 10^{-11}.

Moralité : une grande étude mathématique peut faire gagner énormément de temps...

****

4.12 Table de multiplication (***)

Exercice 21. Écrire une fonction qui affiche la table de Pythagore de la multiplication.

```
    1   2   3   4   5   6   7
 1  1   2   3   4   5   6   7
 2  2   4   6   8  10  12  14
 3  3   6   9  12  15  18  21
 4  4   8  12  16  20  24  28
 5  5  10  15  20  25  30  35
 6  6  12  18  24  30  36  42
 7  7  14  21  28  35  42  49
```

ou mieux

```
+---+---+---+---+---+---+---+---+---+---+---+
|  x|  1|  2|  3|  4|  5|  6|  7|  8|  9| 10|
+---+---+---+---+---+---+---+---+---+---+---+
|  1|  1|  2|  3|  4|  5|  6|  7|  8|  9| 10|
+---+---+---+---+---+---+---+---+---+---+---+
|  2|  2|  4|  6|  8| 10| 12| 14| 16| 18| 20|
+---+---+---+---+---+---+---+---+---+---+---+
|  3|  3|  6|  9| 12| 15| 18| 21| 24| 27| 30|
+---+---+---+---+---+---+---+---+---+---+---+
|  4|  4|  8| 12| 16| 20| 24| 28| 32| 36| 40|
+---+---+---+---+---+---+---+---+---+---+---+
|  5|  5| 10| 15| 20| 25| 30| 35| 40| 45| 50|
+---+---+---+---+---+---+---+---+---+---+---+
|  6|  6| 12| 18| 24| 30| 36| 42| 48| 54| 60|
```

```
+---+---+---+---+---+---+---+---+---+---+---+
|  7|  7| 14| 21| 28| 35| 42| 49| 56| 63| 70|
+---+---+---+---+---+---+---+---+---+---+---+
|  8|  8| 16| 24| 32| 40| 48| 56| 64| 72| 80|
+---+---+---+---+---+---+---+---+---+---+---+
|  9|  9| 18| 27| 36| 45| 54| 63| 72| 81| 90|
+---+---+---+---+---+---+---+---+---+---+---+
| 10| 10| 20| 30| 40| 50| 60| 70| 80| 90|100|
+---+---+---+---+---+---+---+---+---+---+---+
```

Prototype : `void affiche_pythagore(int n);`

Corrigé

```
*****************************************************************************
```

```c
/* afficher une bordure horizontale
+--+--+--+--+
*/
void affiche_bordure(int largeur, int n) {
    int i,j;
    printf("+"); /* le premier '+' */
    for (i=0;i<=n;++i) { /* faire n+1 fois */
        for (j=0;j<largeur;++j) { /* largeur fois '-' */
            printf("-");
        }
        printf("+"); /* plus un '+' */
    }
    printf("\n");
}

/* afficher l'en-tête
| x| 1| 2| 3| 4|
+--+--+--+--+--+
*/
void affiche_en_tete(int largeur, int n) {
    int i;
    printf("|"); /* le premier '|' */
    printf("%*s", largeur, "x"); /* le signe de multiplication sur largeur */
    printf("|"); /* suivi d'un '|' */
    for (i=1;i<=n;++i) { /* pour chaque colonne */
        printf("%*d", largeur, i); /* l'en-tête de colonne sur largeur */
        printf("|"); /* suivi d'un '|' */
    }
    printf("\n");
    affiche_bordure(largeur, n); /* et la bordure basse */
}

/* afficher la ligne i
| 3| 3| 6| 9|12|
+--+--+--+--+--+
*/
void affiche_ligne(int largeur, int i, int n) {
    int j;
    printf("|"); /* le premier '|' */
    printf("%*d", largeur, i); /* l'en-tête de ligne sur largeur */
    printf("|"); /* suivi d'un '|' */
    for (j=1;j<=n;++j) { /* pour chaque colonne */
        printf("%*d", largeur, i*j); /* le produit sur largeur */
        printf("|"); /* suivi d'un '|' */
    }
    printf("\n");
    affiche_bordure(largeur, n); /* et la bordure basse */
```

```
}

/* calcul de la taille d'affichage d'un nombre */
int taille_cellule(int n) {
    int t=1;
    while (n>=10) {
        ++t;
        n/=10;
    }
    return t;
}

/* la fonction d'affichage de la table */
void affiche_pythagore(int n) {
    int ligne;
    int largeur = taille_cellule(n*n); /* taille du plus grand nombre */
    affiche_bordure(largeur, n); /* la bordure du haut */
    affiche_en_tete(largeur, n); /* la ligne d'en-tête */
    for (ligne = 1 ; ligne <= n; ++ligne ) { /* chaque ligne */
        affiche_ligne(largeur, ligne, n);
    }
}
```

****

4.13 Curiosité (********)

Exercice 22. Que fait ce programme ? Discuter de son intérêt.

```
#include <stdio.h>
long a=10000, b, c=840, d, e, f[841], g;
int main () {
    for (;b-c;)
        f[b++]=a/5;
    for (;d=0,g=c*2;c-=14,printf("%.4d",e+d/a),e=d%a)
        for (b=c;d+=f[b]*a,f[b]=d%--g, d/=g--,--b;d*=b);
    return 0;
}
```

Corrigé

****

Il calcule π. Sans commentaire, illisible, incompréhensible, jetable. Bref, à éviter absolument.

****