– Algorithmique – TD N°4 – Procédures & Fonctions & Récursivité

– Corrigé Type –

Exercice 1 : Ordre croissant

Q1) Ecrire une procédure **Trier2** qui prend deux (2) entiers **A** et **B** et de les permuter, si nécessaire, pour que l'état de sortie soit **A ≤ B**. C'est-à-dire que **A** et **B** seront dans l'ordre croissant.

```
Procédure Trier2(Var A : Entier ; Var B : Entier) // Ou Procédure Trier2(Var A, B : Entier)
Var Tmp : Entier        // Tmp est une variable locale
Début
    Si A > B Alors // Si A et B ne sont pas dans l'ordre croissant, il faut les permuter.
        Tmp ← A
        A ← B
        B ← Tmp
    FSi
Fin
```

Q2) En s'inspirant du tri à bulles, écrire une procédure **Trier3** sur trois (3) entiers **(A, B et C)** qui appelle **Trier2** et permet d'avoir les trois variables **A, B et C** dans l'ordre croissant **(A ≤ B ≤ C)**.

```
Procédure Trier3(Var A : Entier ; Var B : Entier; Var C : Entier) // Ou Procédure Trier3(Var A, B, C : Entier)
Début
    Trier2( A, B ) // S'assurer que A et B sont dans l'ordre croissant.
    Trier2( B, C ) // S'assurer que B et C sont dans l'ordre croissant.
    Trier2( A, B ) // S'assurer DE NOUVEAU que A et B sont dans l'ordre croissant.
Fin
```

Q3) Ecrire l'algorithme principal qui permet de lire 3 entiers **(X, Y et Z)** et de les afficher dans l'ordre croissant.

```
Algorithme OrdreCroissant
Var X, Y, Z : Entier   // Trois variables globales
Procédure Trier2(Var A : Entier ; Var B : Entier) // Ou Procédure Trier2(Var A, B : Entier)
Var Tmp : Entier        // Tmp est une variable locale
Début
    Si A > B Alors // Si A et B ne sont pas dans l'ordre croissant, il faut les permuter.
        Tmp ← A
        A ← B
        B ← Tmp
    FSi
Fin
```

Procédure Trier3(**Var** A : Entier ; **Var** B : Entier; **Var** C : Entier) // Ou Procédure Trier3(**Var** A, B, C : Entier)
Début

```
    Trier2( A, B ) // S'assurer que A et B sont dans l'ordre croissant.
    Trier2( B, C ) // S'assurer que B et C sont dans l'ordre croissant.
    Trier2( A, B ) // S'assurer DE NOUVEAU que A et B sont dans l'ordre croissant.
```

Fin
Début

```
    Ecrire("Donner trois entiers X, Y et Z")
    Lire( X, Y, Z )
    Trier3( X, Y, Z )
    Ecrire("Voici ces trois entiers dans l'ordre croissant :")
    Ecrire( X, Y, Z )
```

Fin

Exercice 2 : Carré Parfait

Q1) Ecrire une procédure **CParfait** qui permet de vérifier si un entier positif **N** est un **carré parfait**.
Cette procédure donne deux résultats : un booléen qui est égal à **vrai** si et seulement si **N** est un carré parfait et un entier correspondant à la partie entière de la racine carré de **N**.
Un entier **N** est **carré parfait** s'il est le carré d'un entier **k**, c'est-à-dire $N = k^2$. Par exemple, les entiers 0, 1, 4, 9, 16 et 25 sont des carrés parfaits. Pour chercher la valeur de **k**, on calcule la somme des **k** premiers nombres impairs jusqu'à ce que cette somme soit supérieure ou égale à **N**.

- Si **N = 16** alors S = 1+3+5+7 = la somme des 4 premiers nombres impairs ; **S = N** ➔ 16 est carré parfait et $\sqrt{16}$ **= 4.**
- Si **N = 18** alors S = 1+3+5+7+9 = la somme des 5 premiers nombres impairs ; **S > N** ➔18 n'est pas carré parfait et **la partie entière de $\sqrt{18}$ est égale à 4.**

Procédure CParfait(N : Entier ; **Var** Parfait : Booléen ; **Var** racine : Entier)
Var S, NImpair : Entier // S, NImpair sont des variables locales
Début

```
    S ← 0
    Racine ← 0
    NImpair ← 1  // Prochain nombre impair est le un (1).
    TQ S < N Faire // Vérifier si la somme est supérieure ou égale à N.
        S ← S + NImpair
        NImpair ← NImpair + 2       // Passer au nombre impair suivant
        Racine ← Racine + 1
    FSi
    Si S = N Alors
        Parfait ← Vrai          // N est carré parfait et le calcul de la racine est bon !
```

```
    Sinon
        Parfait ← Faux          // N n'est pas carré parfait
        Racine ← Racine – 1 // Il faut retirer 1 de la partie entière de la racine carrée.
    FSi
Fin
```

Q2) En utilisant la procédure **CParfait**, écrire l'algorithme principal qui permet de : ①**-** Lire un entier positif **N**, ②**-** Vérifier si **N** est un carré parfait et afficher sa racine carrée, sinon afficher la partie entière de sa racine carrée. **Note :** L'utilisation des fonctions SQR (le carré) et SQRT (la racine carrée) n'est pas autorisée.

```
Algorithme CarréParfait
Var     N, R : Entier   // N et R la partie entière de sa racine
        CP : Booléen  // CP ≡ Carré parfait
Procédure CParfait( N : Entier ; Var Parfait : Booléen ; Var racine : Entier)
Var S, NImpair : Entier                  // S, NImpair sont des variables locales
Début
    S ← 0
    Racine ← 0
    NImpair ← 1       // Prochain nombre impair est le un (1).
    TQ S < N Faire // Vérifier si la somme est supérieure ou égale à N.
        S ← S + NImpair
        NImpair ← NImpair + 2   // Passer au nombre impair suivant
        Racine ← Racine + 1
    FSi
    Si S = N Alors
        Parfait ← Vrai     // N est carré parfait et le calcul de la racine est bon !
    Sinon
        Parfait ← Faux    // N n'est pas carré parfait
        Racine ← Racine – 1        // Il faut retirer 1 de la partie entière de la racine carrée.
    FSi
Fin
Début
    Répéter
        Ecrire("Donner un entier positif N ( N ≥ 0 )")
        Lire( N )
    Jusqu'à N ≥ 0
    CParfait( N, CP, R )
    Si CP Alors     // Ou Si CP = Vrai Alors
        Ecrire(N, " est un carré parfait, sa racine = ", R)
    Sinon
        Ecrire(N, " n'est pas un carré parfait, la partie entière de sa racine = ", R)
    FSi
Fin
```

<u>Exercice 3 : Nombres Proniques</u>

Un entier positif **N** est un **nombre pronique** (presque carré) s'il est le produit de deux nombres successifs ***k*** *et* ***(k+1)***, c'est-à-dire ***N = k*(k+1)***. Il est aussi la somme des ***k*** premiers nombres pairs.

Par exemple, les entiers 0 (=0*1) , 2 (=1*2), 6 (=2*3), 12 (=3*4), 20, 30 et 42 sont des nombres proniques.

Pour chercher la valeur de ***k***, on calcule la somme des ***k*** premiers nombres pairs jusqu'à ce que cette somme devienne supérieure ou égale à ***N***.

<u>Exemples</u> : - Si **N = 20** alors S = 2+4+6+8 **= N ➔ N est pronique et k = 4 (20 = 4*5).**

- Si **N = 23** alors S = 2+4+6+8+10 = 30 **> N ➔N n'est pas pronique et k = 5 (23 < 5*6).**

Q1) En respectant cette définition, écrire une fonction **Pronique(N)** qui permet de vérifier si un entier **N** est pronique ou pas.

```
Fonction Pronique( N : Entier) : Booléen
Var     S, I : Entier
Début
        S ← 0
        I ← 2 // Prochain nombre à ajouter
        TQ ( S < N ) Faire      // Tantqu'on a pas dépassé la valeur de N
                S ← S + I
                I ← I + 2
        FTQ
        Si ( S = N ) Alors
                Pronique ← Vrai
        Sinon
                Pronique ← Faux
        FSi
Fin
```

Q2) Ecrire l'algorithme principal qui permet d'afficher les **M** petits nombres proniques.

<u>Exemple</u> : Si M = 10 alors les 10 petits nombres proniques sont : 0, 2, 6, 12, 20, 30, 42, 56, 72, 90.

```
Algorithme NombresProniques
Var     M, N, Cpt : Entier
Fonction Pronique( N : Entier) : Booléen
Var     S, I : Entier
Début
        S ← 0
        I ← 2 // Prochain nombre à ajouter
        TQ ( S < N ) Faire      // Tantqu'on a pas dépassé la valeur de N
                S ← S + I
                I ← I + 2
        FTQ
```

```
    Si ( S = N ) Alors
        Pronique ← Vrai
    Sinon
        Pronique ← Faux
    FSi
Fin
Début
    Répéter
        Ecrire("Donner un entier strictement positif M (M > 0)")
        Lire( M )
    Jusqu'à N > 0
    Cpt ← 0
    N ← 0          // Zéro ( 0 ) est le premier nombre PRONIQUE
    Ecrire("Les ", M, " nombres proniques sont :")
    TQ Cpt < M Faire
        Si Pronique( N ) Alors
            Ecrire( N )
            Cpt ← Cpt + 1
        FSi
        N ← N + 1     // On passe à l'entier suivant
        // On peut améliorer la recherche en incrémentant la valeur de N par 2 car tous les nombres
        // proniques sont pairs. Comme c'est un produit de deux entiers qui se suivent, l'un des deux est
        // forcément pair ; leur produit (multiplication) est pair.
    FTQ
Fin
```

<u>Exercice 4 : Fonctions numériques</u>

Q1) Factorielle : Ecrire une fonction **Facto(n)** qui permet de calculer **n ! = 1 * 2 * 3 * …* n.**

<u>Solution 1</u> :

```
Fonction Facto( n : Entier) : Entier
Var     F, I : Entier
Début
    F ← 1   // 0 ! = 1 et 1 ! = 1
    Pour I ← 1 à N faire     // Ou Pour I ← 2 à N faire car 1 ! est déjà calculé ! Il n'est pas nécessaire de le recalculer
        F ← F * I
    FPour
    Facto ← F
Fin
```

Solution 2 :

```
Fonction Facto( n : Entier) : Entier
Var     F : Entier
Début
        F ← 1   // 0 ! = 1 et 1 ! = 1
        TQ n > 0 faire   // Ou TQ n > 1 faire car 1 ! est déjà calculé ! Il n'est pas nécessaire de le recalculer
                F ← F * I
                n ← n – 1         // n est passé par copie, il retrouvera sa valeur originale à la fin de la fonction.
        FPour
        Facto ← F
Fin
```

Q2) Combinaisons : Ecrire une fonction **Combin(n, p)** qui permet de calculer le nombre de combinaisons défini par $C_n^p = \frac{n!}{p!*(n-p)!}$. **n** et **p** sont deux entiers positifs tels que **p ≤ n**.

```
Fonction Combin( n, p : Entier) : Entier
Début
        Combin ← Facto( n ) Div ( Facto( p ) * Facto( n – p ))       // Utiliser Div au lieu de la division réelle /
Fin
```

Q3) Puissance : Ecrire une fonction **Puiss(x, n)** qui renvoie **x^n = x * x * … * x** (**x** réel et **n** entier positif).

Solution 1 :

```
Fonction Puiss( x : Réel ; n : Entier) : Réel
Var     P, I : Entier
Début
        P ← 1   // x⁰ = 1
        Pour I ← 1 à N faire
                P ← P * x
        FPour
        Puiss ← P
Fin
```

Solution 2 : En s'inspirant de la solution 2 de Facto(n), décrémenter n pour faire le comptage.

Q4) Exponentielle : Ecrire une fonction **Expn(x, n)** qui permet de calculer la valeur approchée $e^x \cong 1+\frac{x}{1}+\frac{x^2}{2!}+\frac{x^3}{3!}+...+\frac{x^n}{n!}$ (**x** est un nombre réel et **n** un nombre entier positif).

```
Fonction Expn( x : Réel ; n : Entier) : Réel
Var     I : Entier
        S : Réel
Début
```

S ← 1 // Le 1[er] terme.

Pour I ← 1 à N faire

S ← S + Puiss(x, I) / Facto(I) // Les termes $\frac{x^i}{i!}$

// Il faut utiliser la division réelle /, mais pas la division entière **Div**

FPour

Expn ← S

Fin

Exercice 5 : Récursivité

Q1) Factorielle : Ecrire une fonction **FactoRec(n)** qui permet de calculer **n ! = n * (n-1) !**

(On sait que 0 ! = 1 ! = 1)

Fonction FactoRec(n : Entier) : Entier

Début

Si (n = 0) OU (n = 1) Alors // Ou Si n ≤ 1 alors

// Cas particuliers 0 ! = 1 et 1 ! = 1

FactoRec ← 1 // un résultat direct

Sinon

// Cas général

FactoRec ← N * FactoRec(n – 1) // Appel récursif

FSi

Fin

Q2) Puissance : Ecrire deux fonctions récursives pour calculer $\mathbf{x^n}$.

- La première **PuissRec(x, n)** en utilisant la définition suivante :

$$x^n = \begin{cases} 1 & Si \quad n=0 \\ x^{n-1} \times x & Si \quad n>0 \end{cases}$$

Fonction PuissRec(x : Réel ; n : Entier) : Réel

Début

Si (n = 0) Alors

// Cas particulier $x^0 = 1$

PuissRec ← 1 // un résultat direct

Sinon

// Cas général

PuissRec ← x * PuissRec (x, n – 1) // Attention : PuissRec a 2 paramètres !

FSi

Fin

- La deuxième **PuissRec2(x, n)** en utilisant la définition suivante :

$$x^n = \begin{cases} 1 & Si\ \ n=0 \\ x^{n/2} \times x^{n/2} & Si\ \ n>0\ \ et\ \ n\ est\ pair \\ x^{(n-1)/2} \times x^{(n-1)/2} \times x & Si\ \ n>0\ \ et\ \ n\ \ est\ impair \end{cases}$$

<u>Solution 1</u> : Réécriture directe

```
Fonction PuissRec2( x : Réel ; n : Entier) : Réel
Début
    Si ( n = 0) Alors
        // Cas particulier x^0 = 1
        PuissRec2 ← 1                  // un résultat direct
    Sinon
        // Cas général
        Si n Mod 2 = 0 Alors           // n est pair
            PuissRec2 ← PuissRec2( x, n Div 2) * PuissRec2( x, n Div 2)
        Sinon
            PuissRec2 ← PuissRec2( x, (n – 1) Div 2) * PuissRec2( x, (n – 1) Div 2) * x
        FSi
    FSi
Fin
```

<u>Solution 1</u> : Solution optimisée

La Solution 1 est correcte et respecte la définition mathématique **MAIS** elle n'exploite pas les faits suivants :

1- PuissRec2(x, n Div 2) est appelée 2 fois avec les mêmes paramètres. Deux appels de fonction pour avoir le même résultat. On peut faire un seul appel et on sauvegarde le résultat obtenu pour l'exploiter 2 fois. On peut écrire par exemple : P ← PuissRec2(x, n Div 2)

Résultat ← P * P

2- Même remarque pour PuissRec2(x, (n – 1) Div 2).

3- L'opération **<u>Div 2</u>** donne le même résultat pour n et (n – 1) si n est impair.

Par exemple, si n = 7 alors n Div 2 = 3 et (n – 1) Div 2 = 3 (n – 1 = 6). Alors, on peut remplacer (n -1) par n. Donc, dans les deux cas pair et impair, on a besoin de calculer une seule fois PuissRec2(x, n Div 2) puis exploiter le résultat obtenu. On peut sortir ce traitement en "facteur commun".

En exploitant ces remarques, on peut réécrire la fonction PuissRec2 comme suit :

```
Fonction PuissRec2( x : Réel ; n : Entier) : Réel
Var     P : Entier
Début
    Si n = 0 Alors
        // Cas particulier x^0 = 1
        PuissRec2 ← 1                  // un résultat direct
```

```
        Sinon
                // Cas général
                P ← PuissRec2( x, n Div 2)   // Un seul appel récursif
                P ← P * P                    // Exploiter le résultat 2 fois
                Si n Mod 2 = 1 Alors         // n est impair
                        P ← P * x            // Multiplier par le dernier x
                FSi
                PuissRec2 ← P
        FSi
Fin
```

Q3) L'algorithme d'Euclide permettant de calculer le **PGCD** (Plus Grand Commun Diviseur) de deux entiers strictement positifs **A** et **B** tels que **A > B** est défini comme suit :

$$PGCD(A,B) = \begin{cases} PGCD(B,\ A \bmod B) & Si\ B \neq 0 \\ A & Si\ B = 0 \end{cases}$$

Ecrire une fonction récursive permettant de déterminer le **PGCD** de **A** et **B**.

```
Fonction PGCD( A, B : Entier) : Entier
Début
        Si B = 0 Alors
                PGCD ← A
        Sinon
                PGCD ← PGCD ( B, A Mod B)
        FSi
Fin
```

Q4) Suite de Fibonacci : Ecrire une fonction récursive **Fibo(n)** permettant de calculer le **n**[ieme] terme de la suite de **Fibonacci** définie par : $U_n = \begin{cases} 0 & Si\ n = 0 \\ 1 & Si\ n = 1 \\ U_{n-1} + U_{n-2} & Si\ n \geq 2 \end{cases}$

```
Fonction Fibo( n : Entier) : Entier
Début
        Si n = 0 Alors
                Fibo ← 0
        Sinon
                Si n = 1 alors
                        Fibo ← 0
                Sinon
```

Fibo ← Fibo (n – 1) + Fibo (n – 2)

FSi

FSi

Fin

Remarque : On peut regrouper les deux cas particuliers comme suit : Si n ≤ 1 Alors Fibo ← n Sinon …

Q5) Pour un entier positif N, écrire une fonction <u>itérative</u> puis une autre <u>récursive</u> qui permet :

- Calculer le nombre de chiffres de N.

```
Fonction NombreChiffres( N : Entier) : Entier
Var     Cpt : Entier
Début
        Cpt ← 1
        TQ N > 9 faire
                N ← N Div 10
                Cpt ← Cpt + 1
        FTQ
        NombreChiffres ← Cpt
Fin
```

// Solution 2

```
Cpt ← 0
Répéter
        N ← N Div 10
        Cpt ← Cpt + 1
Jusqu'à N = 0
```

<u>Version Récursive</u> :

```
Fonction nombreChiffres( N : Entier) : Entier
Début
        Si N ≤ 9 Alors
                nombreChiffres ← 1  // Un seul chiffre
        Sinon
                nombreChiffres ← nombreChiffres( N Div 10 ) + 1 // Retirer un chiffre et compter ceux qui restent.
        FSi
Fin
```

- Calculer la somme des chiffres de N.

```
Fonction sommeChiffres( N : Entier) : Entier
Var     S : Entier
Début
        S ← 0
        TQ N ≠ 0 faire
                S ← S + N Mod 10
                N ← N Div 10
        FTQ
        sommeChiffres ← S
Fin
```

// Solution 2

```
S ← 0
Répéter
        S ← S + N Mod 10
        N ← N Div 10
Jusqu'à N = 0
```

Version Récursive :

```
Fonction sommeChiffres( N : Entier) : Entier
Début
    Si N = 0 Alors
        sommeChiffres ← 0
    Sinon
        sommeChiffres ← N Mod 10 + sommeChiffres( N Div 10 )
    FSi
Fin
```

```
// Solution 2
Si N ≤ 9 Alors
    sommeChiffres ← N  // Un seul chiffre
Sinon
```

- Calculer le produit (la multiplication) des chiffres de N. **Remarque** : Si le nombre N contient un zéro, dès que le chiffre 0 (zéro) est rencontré, le produit devient nul (égal à zéro), il faut s'arrêter.

```
Fonction produitChiffres( N : Entier) : Entier
Var     P : Entier
Début
    P ← N Mod 10
    N ← N Div 10
    TQ ( N ≠ 0 ) ET ( P ≠ 0 ) faire
        P ← P * (N Mod 10)
        N ← N Div 10
    FTQ
    produitChiffres ← P
Fin
```

```
// Solution 2
P ← 1

Répéter
    P ← P * (N Mod 10)
    N ← N Div 10
Jusqu'à ( N = 0 ) OU ( P = 0 )
```

Version Récursive :

```
Fonction produitChiffres( N : Entier) : Entier
Début
    Si N ≤ 9 Alors
        produitChiffres ← N  // Un seul chiffre
    Sinon
        Si N Mod 10 = 0 Alors          // Si un zéro ( 0 ) est rencontré
            produitChiffres ← 0 // Le produit est au final sera nul, il faut s'arrêter !
        Sinon
            produitChiffres ← (N Mod 10 )* produitChiffres( N Div 10 )
        FSi
    FSi
Fin
```