POLYTECH LILLE

Université Lille1 Sciences et Technologies

Département IMA / 3A (S5) **Programmation Structurée 2012/2013**

http://laure.gonnord.org/pro/teaching/

Quick - 2011 - 1h - éléments de correction V2 (modif en 2012)

Exercice 1 Écrire un programme complet qui demande un entier $n > 2$ à l'utilisateur et imprime tous les entiers **pairs** positifs (> 0) et strictement inférieurs à n ($< n$).

CORRECTION : On demande d'écrire un programme complet, on va supposer que l'utilisateur nous renvoie bien un entier > 2. Ensuite, on peut s'apercevoir que l'on peut compter de 2 en 2 jusqu'à $n - 1$ (boucle pour).

```
Programme Main()
    L: n : Entier            {pour stocker l'entier}
    L: i : Entier                {indice de boucle}
    Demander(n)
    Pour i de 2 à n-1 (Pas 2) Faire
        Imprimer i
    Fpour
    Retourner 0
FProgramme
```

On peut par exemple tester avec $n = 7$ et $n = 6$.

Le coût de cet algorithme est de $O(n)$ affichages et additions. ■

Exercice 2 Écrire un programme complet qui demande des entiers positifs ou nuls (≥ 0) à l'utilisateur, s'arrête quand l'utilisateur entre un entier négatif, et imprime à la fin la somme des entiers positifs entrés par l'utilisateur.

CORRECTION : Programme principal. Nous utilisons deux variables locales, x (pour stocker les entrées de l'utilisateur), et s qui va calculer la somme au fur et à mesure. On n'oublie pas d'initialiser s.

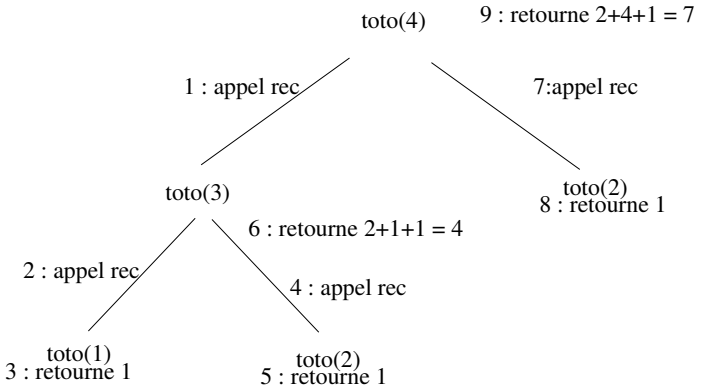
```
Programme Main()
    L: x : Entier        {pour stocker les entiers}
    L: s : Entier           {pour stocker la somme}
    s ← 0
    Demander(x)
    Tq x ≥ 0 faire
        s ← s + x
        Demander(x)
    Ftq
    Imprimer(s)
    Retourner 0
FProgramme
```

Le coût de cet algorithme est $O(n)$ additions où n est le nombre d'entiers entrés par l'utilisateur. ■

Exercice 3 L'algorithme suivant est-il récursif ou itératif ?

Fonction *toto(x) :Entier*
| **D**: x : Entier>0
| **Si** *x=1* **ou** *x=2* **alors**
| | **Retourner** 1
| **Sinon**
| | **Retourner** $2 + toto(x-1) + toto(x-2)$
| **Fsi**
FFonction

Quelles opérations réalise l'appel `toto(4)`, dans quel ordre et quel est le résultat final ?
CORRECTION : Cette fonction est récursive car elle fait appel à elle-même dans sa définition. Ensuite, l'ordre d'exécution peut se représenter par un arbre numéroté :



Attention, les calculs ne se font pas tous à la fin, mais au fur et à mesure des appels récursifs. La calcul de `toto(2)` se fait en particulier 2 fois.

■

Exercice 4 Écrire une **fonction** qui prend un tableau d'entiers (> 0) en paramètre et répond vrai si ce tableau (vecteur) ne contient que des entiers pairs, et faux sinon. (On pourra s'apercevoir que si on trouve un entier impair, on peut terminer l'algorithme...)
CORRECTION : La fonction retourne donc un booléen. Nous allons effectuer un parcours du tableau passé en paramètre, et arrêter dès que l'on trouve un entier impair. Il y a deux solutions (vues en cours), dont voici la plus simple utilisant une boucle pour avec arrêt prématuré.

Fonction *tousPairs(t,N) :Booleen*
| **D**: N : entier {`facultatif`}
| **D**: t :vecteur[N] d'entiers
| **Pour** *i* **de** *0 à N-1* **Faire**
| | **Si** *t[i] mod 2 = 1* **alors**
| | | **Retourner** *Faux* {`il existe un élément impair`}
| | **Fsi**
| **Fpour**
| **Retourner** *Vrai* {`si on est ici c'est qu'on n'a rencontré que des éléments pairs`}
FFonction

Ensuite, on teste que cela fonctionne avec divers tableaux. Le coût de cet algorithme est au pire $N-1$ comparaisons, au mieux 1. ■

Exercice 5 Écrire un algorithme (fonction ou action ? justifier !) qui calcule et stocke dans un tableau (vecteur) de taille N les entiers 0!, 1! $\ldots N-1!$, en faisant un minimum de multiplications. Quelle est la complexité de votre algorithme ?

On rappelle que $i! = 1 \times 2 \times 3 \times \ldots i$.

CORRECTION : On considère $0! = 1$ (non précisé dans l'énoncé). Il y a plusieurs solutions, mais si l'on désire utiliser les calculs à l'extérieur, on a besoin soit de retourner le tableau T (permis en pseudo code mais pas en C), soit de passer le tableau en paramètre résultat (ie par adresse, ce qu'on va faire). Ensuite, on va remplir le tableau en remarquant que $i! = i * (i-1)!$ donc si on stocke $i!$ dans $T[i]$ on peut remplir avec l'instruction $T[i] \leftarrow i * T[i-1]$. Faire attention à l'initialisation et à ne pas accéder à une case qui n'existe pas. ■

Action *remplitfact(T,N)*
| **D**: N : entier {facultatif}
| **D/R**: T :vecteur[N] d'entiers
| $T[0] \leftarrow 1$
| **Pour** *i* **de** *1 à N-1* **Faire**
| | $T[i] \leftarrow i * T[i-1]$
| **Fpour**
FAction

Ensuite on teste pour N=4. La complexité de cet algorithme est $O(n)$ multiplications.