Complément sur la récursivité :

1. Récursivités multiples :

Définition : Un algorithme récursif est multiple si l’un des cas qu’il distingue se résout avec **plusieurs** appels récursifs.

*Exemple de récursivité double* : la suite de Fibonacci : 0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8 ; 13 ; ….. ; …..

*Soit*

*Exercice :*

1. *Implémenter la fonction Fibo1(n) en python qui renvoie le terme de rang n de la suite de Fibonacci*
2. *Tester le temps d’exécution pour n = 10 , 20 , 30 , 40 , 50, …….*
3. *A l’aide d’un schéma, représenter l’exécution de l’appel Fibo(5) :*
4. *Implémenter la fonction Fibo2(n) qui renvoie la liste des termes de la suite de Fibonnacci jusqu’au rang n ( Fibo2(6) doit renvoyer [0,1,1,2,3,5,8]*

*Exercice 2 : Une grenouille doit monter un escalier. Quand elle saute, elle monte de 1 ou 2 marches. Combien de chemins différents existent-il pour un escalier de n marches ? Écrire une fonction récursive donnant la solution*

*Autres exemples de récursivité double : Calcul des coefficients binomiaux ( pour les spé maths)*

1. **Récursivité terminale**

Un algorithme récursif simple est terminal lorsque l’appel récursif est le dernier calcul effectué pour obtenir le résultat. **Il n’y a pas de “calcul en attente”**. L’avantage est qu’il n’y a rien à mémoriser dans la pile.

Une définition possible de fonction f  récursive terminale  est quand tout appel récursif est de la forme **return f(...);** soit un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur.

|  |  |
| --- | --- |
| **Algorithmes non terminal :**  **def** fact(n):  **if** n == 0 :  **return** 1  **else**:  **return** n \* fact(n-1)  **def** occurrences(c,s):  **if** s == "": **return** 0  **elif** c == s[0]:  **return** 1 + occurrences(c,s[1:])  **else**:  **return** occurrences(c,s[1:]) | **Algorithme terminal :**  *Prédicat de présence d’un caractère dans une chaîne :*  Un caractère c ***est présent*** dans une chaîne s non vide, s’il est le premier caractère de s ou s’il ***est présent*** dans les autres caractères de s. Il n’est pas présent dans la chaîne vide.  **def** est\_present(c,s):  **if** s == '':  **return** False  **elif** c == s[0]:  **return** True  **else**:  **return** est\_present(c,s[1:]) |

**Rendre terminal un algorithme récursif**

|  |  |
| --- | --- |
| On utilise un **accumulateur**, passé en paramètre, pour calculer le résultat au fur et à mesure des appels récursifs. La valeur de retour du cas de base devient la valeur initiale de l’accumulateur et lors d’un appel récursif, le “calcul en attente” sert à calculer la valeur suivante de l’accumulateur. | **def** fact\_term (n,acc = 1):  **if** n <= 1:  **return** acc  **else**:  **return** fact\_term(n-1, acc \* n) |

|  |  |
| --- | --- |
| **Exercice 3** : Écrire une version récursive terminale de la fonction occurrences | **Exercice 4** : écrire une version récursive terminale de Fibonacci puis comparer sa rapidité avec la version non terminale. |

### Récursivité croisée

On parle de récursivité croisée lorsque deux fonctions s’appellent l’une l’autre récursivement.

|  |  |
| --- | --- |
| Exemple typique (et très classique) :  **def** pair(n):  **if** n == 0:  **return** True  **else**:  **return** impair(n-1)  **def** impair(n):  **if** n == 0:  **return** False  **else**:  **return** pair(n-1) | **Exercice 2 :** Donner les étapes de l’appel de pair(7) : |