Les fonctions récursives

1. Mécanisme d'appel d'une fonction

Une fonction dans un programme informatique est constituée d'un ensemble d'instruction effectuant un traitement particulier à partir de paramètres fournis lors de l'appel de la fonction.

Dans la mémoire, les fonctions et le programme principal sont stockés à des adresses différentes. L'appel de la fonction par le programme principal a pour effet de déplacer le pointeur programme au début de la fonction afin de l'exécuter. Dans le même temps afin d'assurer la continuité du programme au retour de la fonction, les informations suivantes sont mémorisées dans la pile d'appel (call stack) :

- ✗ Les paramètres de la fonction : Pour mémoriser ces valeurs nécessaires à l'exécution de la fonction
- ✗ Une zone réservée pour la valeur de retour
- ✗ L'adresse de retour afin mémoriser l'adresse de l'instruction du programme principal qui sera exécuté au retour de la fonction
- ✗ Eventuellement d'autres informations suivant le langage utilisé (variable local ...)

Lors de l'appel successif de plusieurs fonctions, chaque appel créé un bloc fonction qui sera empilé sur le bloc de la fonction appelante. Ainsi, l'ordre d'appel des fonctions correspond à l'ordre d'empilement des blocs.

Lorsqu'une fonction est terminée, le bloc est dépilé de la pile afin de récupérer les informations nécessaire à la continuité du programme.

Prenons pour exemple le code suivant :

```
def h(x) :
    return x+1
def g(x) :
    return h(x) + 2 + k(x+1)
def f(x) :
    return g(x) + 1
def k(x) :
    return x+1
```

Q1. **Exécuter** ce programme sur l'IDE Thonny puis **justifier** le résultat renvoyé par l'appel `f(6)`.

On a f(x) = (x+1) + 2 + (x + 1 + 1) + 1

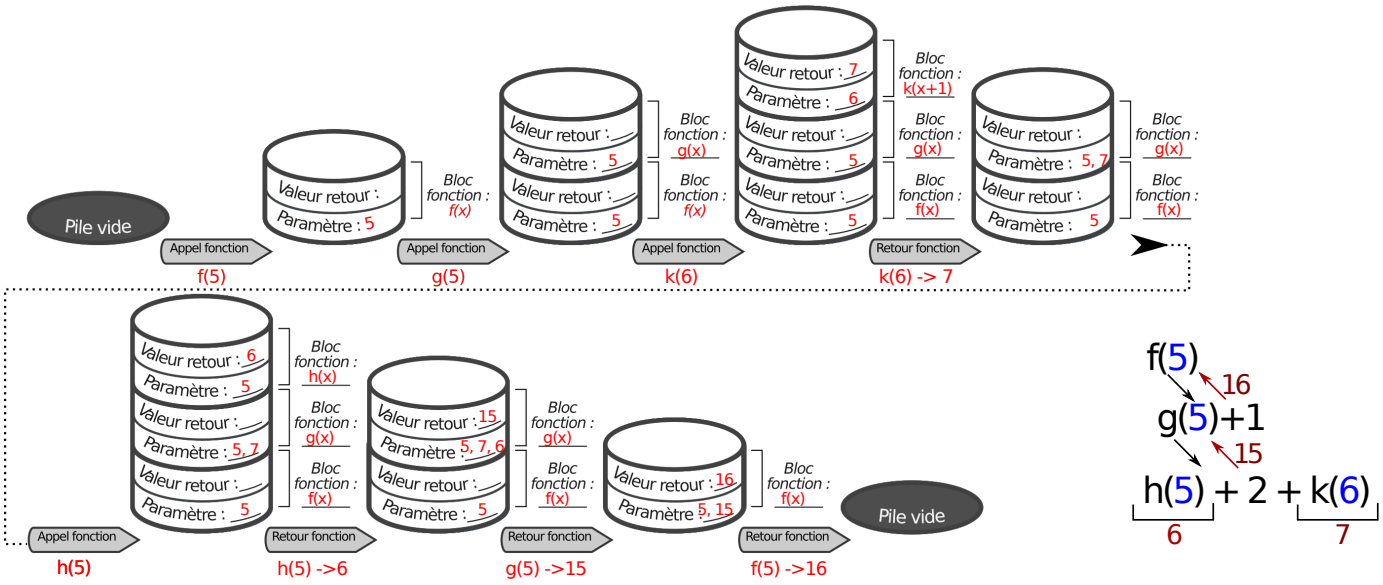
Donc f(6) = 18

Q2. Relever les adresses des différentes fonctions en mémoire. **Comparer** l'ordre d'appel des fonctions lors de l'appel `f(5)`.

Les fonctions ne sont pas adressées dans l'ordre d'appel. Le programme devra donc naviguer dans la mémoire pour exécuter les différents appels puis retourner au fur et à mesure les résultats. Ceci implique de mémoriser la position en mémoire des différentes fonctions appelantes et leur environnement.

La figure 1 représente l'évolution de la pile d'appel lors de l'appel `f(5)`. Elle met en évidence l'empilement successif des blocs fonction lors de appels de fonction ainsi que le dépilement des blocs lors des retours de fonction. Pour simplifier la notation, ces blocs ne font apparaître que les paramètres et les valeurs de retour

Q3. Compléter sur la figure 1 l'ordre d'appel et des retours de fonction. **Compléter** dans chaque bloc la valeur du paramètre transmis ainsi que la valeur de retour au moment ou elles apparaissent lors des appels/retour de fonction.



2.Fonction récursives

Une fonction récursive est une fonction dont le calcul nécessite d'invoquer la fonction elle même.

Prenons pour exemple le calcul de la fonction factorielle (!). On rappel que la factorielle d'un entier n est le produit des nombres entiers strictement positifs et inférieurs ou égal à n.

Pour exemple :

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

On remarque aussi que :

$$5! = 5 \times 4! \text{ avec } 4! = 4 \times 3! \text{ et } 3! = 3 \times 2! \text{ et } 2! = 2 \times 1! \text{ et } 1! = 1$$

Cette dernière expression fait apparaître une fonction récursive puisque la factorielle d'un nombre est égale à ce nombre multiplié par la factorielle de ce nombre moins 1. La figure 2 illustre cette représentation récursive de la fonction factorielle.

Mathématiquement cette fonction récursive s'écrit :

$$n! = n \times (n-1)!$$
$$= (n-1) \times (n-2)!$$
$$= (n-2) \times (n-3)!$$
$$\ldots\ldots\ldots\ldots$$
$$= 2 \times 1!$$
$$= 1$$

Figure 1: Illustration du principe de récursivité dans la fonction factorielle

Cette fonction peut être implémentée en langage Python de la façon suivante :

```
def factorielle(n) :
    if n==1 : return 1
    else : return n*factorielle(n-1)
```

Q4. Justifier que cette fonction est une fonction récursive. **Exécuter** ce programme et **vérifier** le résultat obtenu.

Cette fonction est une fonction récursive car elle s'appelle elle même

Q5. A partir du programme Python de la fonction factorielle, **compléter** sur la figure 2 l'ordre d'appel et des retours des fonctions. **Compléter** dans chaque bloc la valeur du paramètre transmis ainsi que la valeur de retour au moment ou elles apparaissent lors des appels/retour de fonction.

3. Comparaison de la méthode itérative et de la méthode récursives

La même fonction factorielle peut être programmée de manière itérative (avec une boucle for) ou récursive. Les deux programmes sont les suivants :

Version itérative

```
def fact_iterative(n) :
    fact = 1
    for i in range(1,n+1) :
        fact = fact * i
    return fact
```

Version récursive

```
def factorielle(n) :
    if n==1 :
        return 1
    else :
        return n*factorielle(n-1)
```

Q6. Pour les deux méthodes, **déterminer** le nombre de multiplications effectuées pour le cas n !. **Comparer** le coût en temps de ces deux algorithmes.

Le nombre de multiplication est identique dans les deux cas. Le coût en temps est donc identique.

Q7. **Exécuter** les deux programmes pour n = 2000. **Justifier** l'erreur obtenue avec l'algorithme récursif.

Le message d'erreur indique un dépassement (saturation) de la pile d'appel. Le nombre de récursion est donc limité.

Q8. **Conclure** en comparant le coût en temps et le coût en mémoire de ces deux algorithmes.

Les deux algorithmes ont le même coût en temps mais le coût mémoire et bien plus mauvais dans le cas d'un algorithme récursif car l'ordinateur doit conserver en mémoire l'environnement de chaque fonction appelante.

Dans ce cas la fonction récursive est élégante mais moins performante que la solution itérative.