Exponentiation
Notion de récursivité

Christophe Viroulaud

Terminale - NSI

Lang 04

L'*exponentiation* est une opération mathématique définie par :

$$a^n = \underbrace{a \times .... \times a}_{\text{n fois}} \qquad et\ a^0 = 1$$

- $2^4 \rightarrow$ 3 opérations,
- $2701^{103056} \rightarrow$ 103055 opérations.

Comment calculer la puissance d'un nombre de manière optimisée ?

Sommaire

Fonctions Python "built-in"

```python
def puissance_star(x:int,n:int)->int:
    return x**n

def puissance_builtin(x:int,n:int)->int:
    return pow(x,n)
```

Code 1 – Fonctions natives

Activité 1 : Tester les deux fonctions du code 1.

Sommaire

Préconditions

Nous décidons de nous limiter au cas positif.

À retenir

La programmation *défensive* consiste à anticiper les problèmes éventuels.

Activité 2 : Mettre en place un test qui lèvera une `AssertionError` si l'exposant est négatif.

Correction

```
1 def puissance_star(x: int, n: int) -> int:
2     assert n >= 0, "L'exposant doit être positif."
3     return x**n
```

Code 2

Mettre en place des tests

Il existe plusieurs modules (`doctest`) qui facilitent les phases de test.

```
1  import doctest
2
3  def puissance_star(x:int,n:int)->int:
4      """
5      >>> puissance_star(2,8)
6      256
7      >>> puissance_star(2,9)
8      512
9      """
10     return x**n
11
12 doctest.testmod(verbose=True)
```

Code 3 – Tester une fonction

Durée d'exécution

Activité 3 : À l'aide de la bibliothèque `time` mesurer la durée d'exécution de la fonction `puissance_star` pour calculer $2701^{1}9406$.

Correction

```
1 from time import time
2
3 debut=time()
4 puissance_star(2701,19406)
5 fin=time()
6 print("opérande **",fin-debut)
```

Sommaire

S'appuyer sur la définition mathématique

$$a^n = \underbrace{a \times .... \times a}_{n fois} \qquad et \qquad a^0 = 1$$

Activité 4 :

1. Implémenter la fonction **puissance_perso(x : int, n : int)** $\rightarrow$ **int** sans utiliser les fonctions buitin de Python.
2. Mettre en place un test de vérification de la fonction.
3. Mesurer le temps d'exécution de la fonction en l'appelant avec les paramètres (2701,19406).

Correction

```python
def puissance_perso(x:int,n:int)->int:
    """
    >>> puissance_perso(2,8)
    256
    >>> puissance_perso(2,9)
    512
    """
    res = 1
    for i in range(n):
        res*=x
    return res
```

```
opérande ** 0.006058692932128906
fonction pow() 0.005688667297363281
fonction personnelle 0.1307454109191945
```

Code 4 – Les résultats sont significatifs.

Sommaire

Correction de l'algorithme

À retenir

Un **invariant de boucle** est une propriété qui si elle est vraie avant l'exécution d'une itération le demeure après l'exécution de l'itération.

```
1 res = 1
2 for i in range(n):
3     res*=x
```

Code 5 – La propriété $res = x^i$ est un invariant de boucle.

Sommaire

Notation mathématique

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0 \\ x.puissance(x, n-1) & \text{si } n > 0 \end{cases}$$

À retenir

Une fonction **récursive** est une fonction qui s'appelle elle-même.

Sommaire

Implémentation

À retenir

Une fonction récursive :

- s'appelle elle-même,
- possède un **cas limite** pour stopper les appels.

```python
def puissance_recursif(x: int, n: int) -> int:
    if n == 0: # cas limite
        return 1
    else: # appel récursif
        return x*puissance_recursif(x, n-1)
```

Code 6 – Traduction de la formule mathématique

Pile d'appels

```
puissance_recursif(6,4)=
    return 6 * puissance_recursif(6,3)
                    |
                    return 6 * puissance_recursif(6,2)
                                    |
                                    return 6 * puissance_recursif(6,1)
                                                    |
                                                    return 6 * puissance_recursif(6,0)
                                                                    |
                                                                    return 1
```

Visualisation

À retenir

La **pile d'appels** stocke les appels successifs de la fonction récursive.

Remarques

- Python limite la pile d'appels à 1000 récursions.

```
1 import sys
2 sys.setrecursionlimit(20000)
```

Code 7 – Augmenter le nombre de récursions

Remarques

- Python limite la pile d'appels à 1000 récursions.

```
1 import sys
2 sys.setrecursionlimit(20000)
```

Code 8 – Augmenter le nombre de récursions

- La durée d'exécution ne s'est pas améliorée.

```
1 fonction récursive
     0.16802310943603516
```

Sommaire

Nouvelle formulation mathématique

$$a^{2048} = \Bigg(\Bigg(\Bigg(\Bigg(\Bigg(\Bigg(\Bigg(\Bigg(\Bigg(\Bigg((a^2)^2 \Bigg)^2 \Bigg)^2 \Bigg)^2 \Bigg)^2 \Bigg)^2 \Bigg)^2 \Bigg)^2 \Bigg)^2 \Bigg)^2 .$$

FIGURE 1 – Exponentiation rapide

$$puissance(x, n) =$$

$$\begin{cases} 1 & \text{si } n = 0 \\ puissance(x * x, n/2) & \text{si } n > 0 \text{ et n pair} \\ x.puissance(x * x, (n-1)/2) & \text{si } n > 0 \text{ et n impair} \end{cases}$$

Activité 5 : Implémenter la fonction `puissance_recursif_rapide(x: int, n: int)→ int` qui traduit la formulation récursive précédente.

Correction

```
1  def puissance_recursif_rapide(x: int, n: int) ->
       int:
2      if n == 0: # cas limite
3          return 1
4      elif n % 2 == 0: # pair
5          return puissance_recursif_rapide(x*x, n//2)
6      else: # impair
7          return x*puissance_recursif_rapide(x*x, n
       //2)
```

Code 9 – Exponentiation rapide

Visualisation

```
fonction récursive rapide
   0.021007537841796875
```

Code 10 – Les résultats s'améliorent sans égaler la fonction native.

itératif plus rapide car appels fonction coûtent ; mais récursif donne souvent code plus clair/lisible