Exponentiation
Notion de récursivité
Christophe Viroulaud
Terminale - NSI

Exponentiation
Notion de récursivité

Christophe Viroulaud

Terminale - NSI

L'*exponentiation* est une opération mathématique définie par :
$a^n = \underbrace{a \times .... \times a}_{\text{n fois}}$ et $a^0 = 1$

Un calcul comme 3^4 ne pose pas de problème mais 2701^{103056} peut prendre un certain à effectuer par le langage de programmation.

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

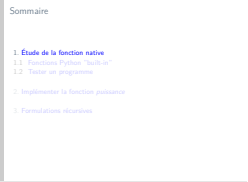
$$a^n = \underbrace{a \times .... \times a}_{\text{n fois}} \qquad \text{et } a^0 = 1$$

▶ $2^4 \rightarrow$ 3 opérations,
▶ $2701^{103056} \rightarrow$ 103055 opérations.
Comment calculer la puissance d'un nombre de manière optimisée ?

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

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

Sommaire



Sommaire

Fonctions Python "built-in"

```
1 def puissance_star(x:int,n:int)->int:
2     return x**n
3
4 def puissance_builtin(x:int,n:int)->int:
5     return pow(x,n)
```

Code 1 – Fonctions natives

Activité 1 : Tester les deux fonctions du code 1.

Fonctions Python "built-in"

```
1 def puissance_star(x:int,n:int)->int:
2     return x**n
3
4 def puissance_builtin(x:int,n:int)->int:
5     return pow(x,n)
```

Code 1 – Fonctions natives

Activité 1 : Tester les deux fonctions du code 1.

Sommaire

1. Étude de la fonction native
1.1 Fonctions Python "built-in"
1.2 Tester un programme
Préconditions
Mettre en place des tests
Durée d'exécution
2. Implémenter la fonction puissance
3. Formulations récursives

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.

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

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

Code 2

Correction

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

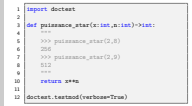
Code 2

Mettre en place des tests

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

Mettre en place des tests

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



Code 3 – Tester une fonction

```python
import doctest

def puissance_star(x:int,n:int)->int:
    """
    >>> puissance_star(2,8)
    256
    >>> puissance_star(2,9)
    512
    """
    return x**n

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 $2701^{1}9406$.

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

Correction

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

Sommaire

Sommaire

S'appuyer sur la définition mathématique

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

Activité 4 :

1. Implémenter la fonction **puissance_perso(x : int, n : int) → 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).

S'appuyer sur la définition mathématique

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

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

```
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 0.13074541091918945
```

Code 4 – Les résultats sont significatifs.

Correction

```
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 0.13074541091918945
```

Code 4 – Les résultats sont significatifs.

Sommaire

1. Étude de la fonction native
2. Implémenter la fonction *puissance*
2.2 Correction de l'algorithme
3. Formulations récursives

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.

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.

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

C'est en fait un raisonnement par récurrence comme en mathématiques.

```
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
1. Étude de la fonction native
2. Implémenter la fonction puissance
3. Formulations récursives
3.1 Notation mathématique
3.2 Implémentation
3.3 Nouvelle formulation mathématique

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.

récursivité = technique de programmation // impératif

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

Sommaire

Implémentation

À retenir

Une fonction récursive :

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

Code 6 – Traduction de la formule mathématique


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

Code 6 – Traduction de la formule mathématique

Pile d'appels

Visualisation

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.

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

1. Il n'y a pas de raison que ça soit mieux : le nombre d'opérations reste le même
2. même un peu moins bien : la récursivité est moins bien géré par l'interpréteur Python que par d'autres langages (Ocaml)

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

1. Il n'y a pas de raison que ça soit mieux : le nombre d'opérations reste le même
2. même un peu moins bien : la récursivité est moins bien géré par l'interpréteur Python que par d'autres langages (Ocaml)

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

Sommaire

Nouvelle formulation mathématique

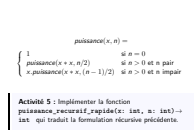
$$a^{2048} = \left(\left(\left(\left(\left(\left(\left(\left(\left(a^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2.$$

FIGURE 1 – Exponentiation rapide

Nouvelle formulation mathématique

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

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.

$$puissance(x, n) =$$

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

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

Correction

Code 9 – Exponentiation rapide

Visualisation

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

```
1 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

1. Implémentation des fonctions builtin en C
2. '

```
1 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