

# 1 Problématique

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

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

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

## 2 Étude de la fonction native

### 2.1 Fonctions Python "built-in"

fournies par Python et/ou langages de haut-niveau.

```
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)
```

### 2.2 Tester un programme

#### 2.2.1 Préconditions

Nous nous limitons au cas positif.

évoquer la *programmation défensive*

■ **Activité 1** : Mettre en place un test qui lèvera une *AssertionError* si l'exposant est négatif.

#### 2.2.2 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)
```

## 2.3 Temps d'exécution

Observons la durée d'exécution de nos fonctions, pour de grandes valeurs de paramètres.

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

## 3 Implémenter la fonction *puissance*

### 3.1 S'appuyer sur la définition mathématique

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

#### Activité 2 :

1. Implémenter la fonction *puissance\_perso*(*x:int,exp:int*)->*int* sans utiliser les fonctions builtin 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).

```
1 def puissance_perso(x:int,n:int)->int:
2     """
3     >>> puissance_perso(2,8)
4     256
5     >>> puissance_perso(2,9)
6     512
7     """
8     res = 1
9     for i in range(n):
10         res*=x
11     return res
```

### 3.2 Invariant de boucle

Il permet de prouver la *correction* d'un algorithme.

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

La propriété  $res = x^i$  est un invariant de boucle. C'est en fait un raisonnement par récurrence comme en mathématiques.

### 3.3 Temps d'exécution

durée ×20 par rapport aux fonctions builtin. Qu'en conclure ?



pourquoi n'obtient-on pas encore une durée similaire aux fonctions builtin ?

— Implémentation des fonctions builtin en C

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

— python pas optimisé pour récursif

code Python est opensource.

on peut passer impératif <-> récursif

```
1 def puissance_iteratif_rapide(x,n):
2     res=1
3     while n>0:
4         if n % 2 == 0:
5             x = x*x
6             n = n // 2
7         else:
8             res = res * x
9             n = n - 1
10    return res
```