

## Diviser pour régner

### Introduction

Pour résoudre un problème de taille N, **un algorithme récursif** fonctionne en général de la manière suivante :

- **Extraire ou construire** à partir de notre entrée un problème de taille N-1
- **Résoudre** le problème de taille N-1 (récursivité)
- **Utiliser cette solution** pour résoudre le problème initial

La méthode **diviser pour régner** consiste, pour résoudre un problème de taille N, à :

(Diviser) : **partager le problème en sous-problèmes** (par exemple de taille N/2)

(Régner) : **résoudre ces différents sous-problèmes** (généralement récursivement)

(Combiner) : **fusionner les solutions** pour obtenir la solution du problème initial

On obtient en général moins d'appels récursifs. Dans certains cas la méthode diviser pour régner donne un algorithme de résolution plus rapide.

### Exponentiation rapide

On peut définir  $x^n$  de **façon récursive** :

$$x^0 = 1$$

$$x^n = x * x^{n-1}$$

Ce qui se traduit en Python par cette fonction :

```
def expo_rec(x,n):  
    if n < 1:  
        return 1  
    else:  
        return x*expo_rec(x,n-1)
```

La **complexité** (en regardant le nombre de multiplication) de cet algorithme est en  $O(n)$ , il faudra donc faire 100 multiplications pour calculer  $x^{100}$ .

On peut mesurer le **temps d'exécution** en utilisant la bibliothèque **timeit** :

```
from timeit import default_timer as timer  
def expo_rec(x,n):  
    if n < 1:  
        return 1  
    else:  
        return x*expo_rec(x,n-1)  
  
d=timer()  
print(expo_rec(2,100))  
f=timer()  
print(f-d)
```

**Exercice 1:** Écrire ce programme.

À partir de quelle valeur de n dépasse on les capacités de la pile ?

---

## Une autre approche (Diviser pour régner)

---

On peut définir  $x^n$  d'une autre façon :

$$x^0 = 1$$

$$\text{Si } n \text{ est pair : } x^n = (x^{n/2})^2$$

$$\text{Si } n \text{ est impair : } x^n = x \cdot (x^{(n-1)/2})^2$$

Ce qui se traduit en Python par cette fonction :

```
def expo_dr(x,n):
    if n<1:
        return 1
    else:
        if n%2==0:
            return expo_dr(x*x,n//2)
        else:
            return expo_dr(x*x,n//2)*x
```

La **complexité** (en regardant le nombre de multiplications) de cet algorithme est en  $O(\log_2(n))$ , il faudra donc faire 7 multiplications pour calculer  $x^{128}$ .

**Exercice 2:** Écrire ce programme.

Vérifier que la différence des temps d'exécutions des deux programmes est évidente pour par exemple le calcul de  $2^{500}$

Vérifier également que le second programme permet de calculer  $2^{1000}$

---

*Une vidéo pour expliquer les complexités*

---

## Complexité et Récursion

Intervenant: **Christian QUEINNEC**

Professeur Émérite, Université Pierre et Marie Curie (Paris VI)