# Étude de la complexité temporelle des algorithmes de calcul de $x^n$

TD-P#1 – Remise à niveau pour l'informatique

## 1 Introduction

Le calcul de $x^n$ est une opération fondamentale en informatique et en mathématiques. Plusieurs algorithmes peuvent être utilisés pour effectuer cette opération, chacun ayant des complexités différentes. Ce TP propose de comparer les complexités temporelles de trois algorithmes de calcul de $x^n$ : l'algorithme naïf, l'algorithme par exponentiation rapide méthode récursive et l'algorithme par exponentiation rapide itérative. Les codes, résultats, discussions et conclusions seront présentés dans un rapport LaTeX, Python note book ou autre.

## 2 Algorithmes

### 2.1 Algorithme Naïf

L'algorithme naïf calcule $x^n$ en multipliant $x$ par lui-même $n$ fois.

$$x^n = \underbrace{x \times \cdots \times x}_{n \text{ fois}}$$

Questions 1.

1. Écrire la fonction `naive_pow(x,n)` qui implante la méthode naïve du calcul de $x^n$

2. Compter le nombre de multiplications. Exprimer ce résultat avec un ordre de grandeur adapté.

### 2.2 Exponentiation Rapide

L'exponentiation rapide, ou exponentiation binaire, divise le problème en deux à chaque étape.

Par exemple, en base 2, $n = \sum_{k=0}^{d} a_k 2^k$ pour $a_k \in \{0, 1\}$. Donc,

$$x^n = x^{a_0}(x^2)^{a_1}(x^{2^2})^{a_2} \dots (x^{2^d})^{a_d}$$

Questions 2.

1. Quelle opération est réalisée pour calculer les $x^{2^k}$ successifs et combien de fois ?

2. Compter le nombre de produit des $(x^{2^k})^{a_k}$.

3. En déduire le nombre d'opérations en fonction de leur nature pour calculer $x^n$.

Soit $n$ un entier strictement supérieur à 1, supposons que l'on sache calculer, pour chaque réel $x$, toutes les puissances $x^k$ de $x$, pour tout $k$, tel que $1 \leq k < n$.

- Si $n$ est pair alors $x^n = (x^2)^{n/2}$. Il suffit alors de calculer $y^{n/2}$ pour $y = x^2$.
- Si $n$ est impair et $n > 1$, alors $x^n = x(x^2)^{(n-1)/2}$. Il suffit de calculer $y^{(n-1)/2}$ pour $y = x^2$ et de multiplier le résultat par $x$.

#### 2.2.1 Exponentiation Rapide Récursive:"Square and Multiply"

On en déduit l'algorithme récursif suivant qui calcule $x^n$ pour un entier strictement positif $n$ :

$$\texttt{puissance}(x,n) = \begin{cases} x, & \text{si } n = 1 \\ \texttt{puissance}(x^2, n/2), & \text{si } n \text{ est pair} \\ x \times \texttt{puissance}(x^2, (n-1)/2), & \text{si } n \text{ est impair} \end{cases}$$

**Questions 3.**

1. Écrire une fonction `SquareMultiplyRecursive(x,n)` qui calcule et retourne $x^n$ avec la méthode récursive précédente.
2. Compter `S` et `M` les nombres respectifs de carrés (Square) et de multiplications (Multiply) pour une instance de la fonction.
3. Écrire une fonction `BitsCount(n)` qui retourne le nombre de chiffres dans la décomposition en base 2 d'un entier $n$.
4. Écrire une fonction `weight(n)` qui retourne le nombre de bits à 1 dans la décomposition en base 2 d'un entier $n$.
5. Déterminer le lien entre `S, M` et les résultats des fonctions précédentes.
6. Vérifier que $n = (n_d \dots n_1 n_0)_2$ pour $T(n_d \dots n_1 n_0) = S + n_0 M + T(n_d \dots n_1)$ après un appel récurif. Donner l'expression de $T(n_d \dots n_1 n_0)$ à la fin del'algorithme récursif . `SquareMultiplyRecursive`.

### 2.2.2 Exponentiation Rapide Itérative :"Square and Multiply"

C'est une méthode itérative pour calculer $x^n$ avec l'exponentation rapide.

**Questions 4.**

1. Estimer le nombre de passage dans la boucle dans l'algorithme itératif, lister les opérations et désigner les opérations significatives.
2. Décrire les instances de l'algorithme pour le meilleur des cas et le pire des cas.
3. Écrire une fonction `SquareMultiplyIterative(x,n)` qui calcule et retourne $x^n$ avec la méthode itérative décrite dans l'algorithme.
4. Compter `S` et `M` le nombre respectifs de carrés (Square) et de multiplications (Multiply) pour une instance de la fonction itérative.
5. Déterminer le lien entre `S, M` et les résultats des fonctions `BitsCount` et `weight`.
6. En déduire, le nombre déduire le nombre de carrés et de multiplication. `SquareMultiplyIterative`.
7. Proposer aussi une preuve formelle pour le calcul du nombre de carrés et de multiplications de l'algorithme.

```
1  ALGORITHME SquareMultiplyIterative(x, n)
2  DONNEES x reel, n entier
3  VARIABLES result, base des reels
4  DEBUT
5      result ← 1
6      base ← x
7      TQ n > 0 FAIRE
8          SI n est impair ALORS
9              result ← result * base
10         base ← base * base
11         n ← ⌊ n/2 ⌋
12     RENVOYER result
13 FIN
```

# 3 Comparaison des Performances

Pour comparer les performances des différents algorithmes, on propose de mesurer le temps d'exécution pour des valeurs croissantes de $n$ en utlisant les bibliothèques `time` et `matplotlib`, de la façon suivante :

Listing 1: Comparaison des performances

```
import time
import matplotlib.pyplot as plt

def measure_time(func, x, n):
```

```
    start_time = time.time()
    func(x, n)
    return time.time() - start_time

x = 2
n_values = [i for i in range(1, 100000,1000)]
times_naive = []
times_recursive = []
times_iterative = []

for n in n_values:
    times_naive.append(measure_time(naive_pow, x, n))
    times_recursive.append(
          measure_time(SquareMultiplyRecursive, x, n))
    times_iterative.append(
          measure_time(SquareMultiplyIterative, x, n))

plt.plot(n_values, times_naive, label='Naif')
plt.plot(n_values, times_recursive, label='Recursive')
plt.plot(n_values, times_iterative, label='Iterative')
plt.xlabel('n')
plt.ylabel('Temps (secondes)')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.title('Comparaison des Algorithmes de Calcul de x^n')
plt.show()
```

Questions 5.

1. Comparer les trois méthodes algorithmes en traçant des courbes pour les valeurs de $n$ données dans le listing.
2. Éliminer la comparaison avec la méthode naïve et l'affichage des échelles logarithmiques. Commenter les nouvelles courbes.
3. Modifier l'algorithme itératif pour éliminer le dernier carré (n=0) très coûteux. Et comparer de nouveaux les courbes. Commenter.
4. Tracer les courbes pour des grandes valeurs de $n$ dans le meilleur et le pire des cas.
5. Comparer les temps de calculs avec celui de l'opérateur de Python `x**n`
6. Ajouter les courbes de $C * n\sqrt{n}$ et $K * n^{\log_2 3}$ en ajustant les constantes pour les valeurs de $n$ dans le pire des cas.
7. Discuter et commenter les résultats obtenus sur la complexité temporelle de l'exponentation rapide.

# 4 Rapport

Présenter les codes,les courbes, les résultats, les discussions et conclusions dans un rapport LaTeX, Python Notebook ou autre.