Calcul Formel - Master 1 - Univ. Paris-Saclay ①

Chapitre 1 - Partie 1

I Ordres de grandeur.

Notations: O, $\sim$. Si $P \in \mathbb{R}[X]$ de degré d alors $P(n) \sim a_n n^d$, d'où $P(n) = O(n^d)$.

Alors $\sum_{k=0}^{n} P(k) = O(n^{d+1})$.

Th principal (§1.3.2) : Soit (u_n) une suite de réels strictement positifs. Alors:

(i) Si $u_{n+1} \leq \lambda u_n$ pour tout n alors $u_n = O(\lambda^n)$. (ici $\lambda > 0$ fixé)

(ii) Si $u_{n+1} \leq u_n + O(n^k)$ alors $u_n = O(n^{k+1})$ (ici $k \geq 0$ fixé).

(iii) Si $u_n \leq u_{\lceil n/2 \rceil} + O(1)$ alors $u_n = O(\log_2 n)$.

(iv) Si $u_n \leq 2 u_{\lceil n/2 \rceil} + O(n)$ alors $u_n = O(n \log_2 n)$.

NB : $u_n = O(v_n) \iff \exists c > 0 \quad \forall n \in \mathbb{N}^* \quad u_n \leq c v_n$

Si $v_n > 0$ pour tout $n \in \mathbb{N}^*$: $u_n = O(v_n) \iff \exists c > 0$ $\forall n$ assez grand $u_n \leq c v_n$.

⚠ si $v_n = \deg P$, on écrira souvent $O(\underbrace{\deg P + 1}_{>0})$ pour tout $P \neq 0$ (même si P est constant).

II Exponentiation Rapide.

A anneau, $a \in A$, $n \in \mathbb{N}$. On veut calculer a^n.
A ou G groupe noté multiplicativement.

1) Exponentiation naïve.

```
def expo_naive(a,n):
    if n==0:
        return 1
    else: return a * expo_naive(a,n-1)
```

⊛ Notons u_n le nombre de multiplications utilisées pour calculer a^n par cet algorithme.
Alors: $u_n = u_{n-1} + O(1)$ donc $u_n = O(n)$.

⊛ Complexité en espace / mémoire : capacité de stockage utilisée. Ici $O(n)$ entiers à stocker.
Itératif au lieu de récursif : $O(1)$

2) Exponentiation rapide.

Idée: $x^{16} = x^{2\times 8} = (x^8)^2 = \left((x^4)^2\right)^2 = \left(\left((x^2)^2\right)^2\right)^2$ donc on peut calculer x^{16} en 4 multiplications.

Exponentiation rapide pour calculer a^n :
→ Si $n = 0$ ou 1 évident
→ Si $n \geq 2$ calculer $y = a^{\lfloor n/2 \rfloor}$; si n pair : $a^n = y^2$
si n impair : $a^n = y^2 * y$.

Notons u_n le nombre de multiplications pour calculer ainsi a^n : on a $u_n \leq u_{\lfloor n/2 \rfloor} + O(1) \leq u_{\lceil n/2 \rceil} + O(1)$

donc : **Th : $u_n = O(\log_2 n)$**.

Coût en mémoire : implémentation récursive : $O(\log_2 n)$; itérative : $O(1)$.

Chapitre 1 - Partie 2

II Algorithme de Karatsuba.

$x = \sum_{i=0}^{n-1} a_i 2^i$ avec $a_i \in \{0,1\}$. On dit que a_i est un bit et que x est un entier à n bits.

On a : $0 \leq x \leq 2^n - 1$.

Th (Karatsuba) : On peut multiplier deux entiers à n bits en $O(n^{\log_2 3})$ opérations élémentaires sur les bits, avec un coût en mémoire $O(n)$ bits. NB : $\log_2 3 \approx 1{,}58$.

Preuve : On peut supposer n pair (et même que n est une puissance de 2 : il existe toujours $n \leq n' \leq 2n$ tel que n' soit une puissance de 2, $O(n'^{\log_2 3}) = O(n^{\log_2 3})$).

Notons $n = 2m$, $x = 2^m a + b$, $y = 2^m c + d$ avec a, b, c, d entiers à m bits.

$$xy = 2^{2m} ac + 2^m \underbrace{(ad + bc)}_{= N - ac - bd} + bd$$

Posons $N = (a+b)(c+d) = ac + bd + (ad+bc)$

Idée : calculer ac, bd et N : nécessite 3 produits d'entiers à m bits
faire $O(1)$ sommes entre entiers à m ou n ou $2n$ bits : coûte $O(n)$ opérations sur les bits.

Notons $m = 2^k$, $n = 2^{k+1}$, et c_k le coût en opérations élémentaires sur les bits du produit de 2 entiers à 2^k bits. Alors : $c_{k+1} \leq 3c_k + O(2^k)$

Posons $d_k = \frac{c_k}{3^k}$; alors $d_{k+1} \leq d_k + O\left(\left(\frac{2}{3}\right)^k\right)$ donc $0 \leq d_{k+1} - d_k = O\left(\left(\frac{2}{3}\right)^k\right)$ donc $\sum d_{k+1} - d_k$ CV et (d_k) CV.

Donc $d_k = O(1)$ et $c_k = O(3^k)$ avec $k+1 = \log_2 n$ donc $3^{k+1} = 3^{\log_2 n} = e^{(\log_2 n)(\log 3)} = 2^{(\log_2 n)\frac{\log 3}{\log 2}}$
$= n^{\frac{\log 3}{\log 2}}$.

Coût en espace : notons M_k le coût pour multiplier deux entiers à 2^k bits.

Alors $M_{k+1} = M_k + O(2^k)$ donc $M_{k+1} - M_k \leq \lambda 2^k$ avec $\lambda > 0$.

Donc $M_k \leq \lambda \left(\sum_{i=0}^{k} 2^i \right) + M_0 \leq M_0 + \lambda 2^{k+1} = O(2^k)$ donc coût en $O(n)$.

Th (Karatsuba sur des polynômes) : Soit A un anneau commutatif.

Alors on peut multiplier deux polynômes à coefficients dans A de degré $\leq n$ en $O(n^{\log_2 3})$ opérations arithmétiques dans A, avec un coût en espace $O(n)$.

↳ sommes, produits, soustractions.

⚠ différence essentielle entre opérations sur entiers / polynômes : retenues.

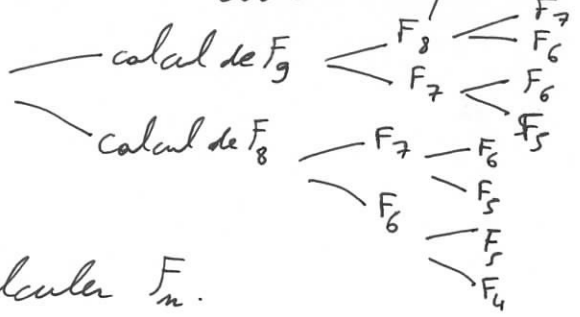
$$\left(\sum_{i=0}^{n-1} a_i X^i \right) \left(\sum_{j=0}^{n-1} b_j X^j \right) = \sum_{k=0}^{2n-2} \underbrace{\left(\sum_{i+j=k} a_i b_j \right)}_{\text{coefficient de } X^k} X^k \qquad a_i, b_j \in A$$

Chapitre 1 - Partie 3

III Suite de Fibonacci. $F_0 = 0$, $F_1 = 1$, $F_{n+2} = F_n + F_{n+1}$ pour tout $n \in \mathbb{N}$.

⊛ Expression explicite de F_n : $F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$. Coût: $O(\log_2 n)$ opérations arithmétiques dans $\mathbb{Q}(\sqrt{5})$.

⊛ Calcul récursif : temps de calcul explose ! Calcul de F_{10} :



Mémoïsation : stocker dans un tableau les valeurs de F_n déjà calculées.

$\rightsquigarrow$ [coût en nombre d'opérations en $O(n)$ pour calculer F_n.
coût en mémoire $O(n)$

⊛ Calcul itératif : [coût $O(n)$ opérations arithmétiques. $(F_n, F_{n+1}) \rightsquigarrow (F_{n+1}, F_{n+2} = F_n + F_{n+1})$
coût en mémoire $O(1)$

⊛ Cas particulier de suites récurrentes linéaires : posons $X_n = \begin{bmatrix} u_n \\ u_{n+1} \end{bmatrix}$ et $M = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$. Alors :

$$\boxed{X_{n+1} = M X_n} \quad \text{car} \quad \begin{cases} u_{n+1} = u_{n+1} \\ u_{n+2} = u_n + u_{n+1} \end{cases} \quad \text{D'où} \quad \boxed{X_n = M^n X_0 = M^n \begin{bmatrix} u_0 \\ u_1 \end{bmatrix}}$$

Calcul de M^n par exp. rapide: coût $O(\log_2 n)$ multiplications de matrices de $M_2(\mathbb{Z})$
$= O(\log_2 n)$ opérations arithmétiques dans $\mathbb{Z}$.

Coût du calcul de F_n: $O(\log_2 n)$ opérations arith. dans $\mathbb{Z}$.