Notions de complexité

Vanessa VITSE

Université Grenoble Alpes

Prépa Agreg Option C 2023-2024

Section 1

Introduction à la complexité

Introduction à la complexité

But : mesurer les ressources nécessaires à l'exécution d'un algorithme

- temps (le plus souvent)
- mémoire (parfois)

Introduction à la complexité

But : mesurer les ressources nécessaires à l'exécution d'un algorithme

- temps (le plus souvent)
- mémoire (parfois)

Premier exemple

Calcul des termes de la suite de Fibonacci

$$F_{k+2} = F_{k+1} + F_k \bmod n$$

avec $F_0 = 1,\ F_1 = 1,\ n$ entier quelconque

$\rightarrow$ on compte le nombre d'opérations (addition/multiplication) dans $\mathbb{Z}/n\mathbb{Z}$
$C(k) =$ nombre d'opérations dans $\mathbb{Z}/n\mathbb{Z}$ pour calculer F(k)

La suite de Fibonacci

Premier programme récursif

```
A=Zmod(101)
def F(k):
    if k==0 or k==1:
        return A(1)
    else:
        return F(k-1)+F(k-2)
```

La suite de Fibonacci

Premier programme récursif

```
A=Zmod(101)
def F(k):
    if k==0 or k==1:
        return A(1)
    else:
        return F(k-1)+F(k-2)
```

$C(k) = C(k-1) + C(k-2) + 1,\ C(0) = C(1) = 0$
donc $C(k) = F(k) - 1 = O(\varphi^k)$ opérations dans $\mathbb{Z}/n\mathbb{Z}$ avec $\varphi = \frac{1+\sqrt{5}}{2}$

La suite de Fibonacci

Deuxième programme itératif

```
def F2(k):
    F0=A(1)
    F1=A(1)
    for i in range(2,k+1):
        F1,F0=F1+F0,F1
    return F1
```

La suite de Fibonacci

Deuxième programme itératif

```
def F2(k):
    F0=A(1)
    F1=A(1)
    for i in range(2,k+1):
        F1,F0=F1+F0,F1
    return F1
```

$C(k) = O(k)$ opérations dans $\mathbb{Z}/n\mathbb{Z}$

La suite de Fibonacci

On exploite le fait que

$$\begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_k \\ F_{k-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

La suite de Fibonacci

On exploite le fait que

$$\begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_k \\ F_{k-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

On calcule $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k$ dans $\mathbb{Z}/n\mathbb{Z}$ par exponentiation rapide

La suite de Fibonacci

On exploite le fait que

$$\begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_k \\ F_{k-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

On calcule $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k$ dans $\mathbb{Z}/n\mathbb{Z}$ par exponentiation rapide

Troisième programme

```
def F3(k):
    M=(matrix([[A(1),A(1)],[A(1),A(0)]]))^k
    return M[1,0]+M[1,1]
```

La suite de Fibonacci

On exploite le fait que

$$\begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_k \\ F_{k-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

On calcule $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k$ dans $\mathbb{Z}/n\mathbb{Z}$ par exponentiation rapide

Troisième programme

```
def F3(k):
    M=(matrix([[A(1),A(1)],[A(1),A(0)]]))^k
    return M[1,0]+M[1,1]
```

$C(k) = O(\log(k))$ mult matricielle $= O(\log(k))$ opérations dans $\mathbb{Z}/n\mathbb{Z}$
(1 mult matricielle $=$ 8 mult $+$ 4 add dans $\mathbb{Z}/n\mathbb{Z}$)

Modèle de calcul

Tps d'exécution du calcul intéressant mais dépend de plein de facteurs : machine utilisée, langage, puissance, gestion mémoire...

- on compte le nb d'opérations élémentaires effectuées par l'ordinateur et la taille des objets utilisés
- préciser les opérations élémentaires en début d'analyse

Modèle de calcul

Tps d'exécution du calcul intéressant mais dépend de plein de facteurs : machine utilisée, langage, puissance, gestion mémoire...

- on compte le nb d'opérations élémentaires effectuées par l'ordinateur et la taille des objets utilisés
- préciser les opérations élémentaires en début d'analyse

Exemple : calcul du pgcd de deux entiers avec Euclide

- op. élém. = division euclidienne
- raffinement : chaque division fait intervenir des entiers de plus en plus petits $\rightsquigarrow$ à prendre en compte pour plus de précision dans l'analyse de la complexité

Modèle de calcul

Tps d'exécution du calcul intéressant mais dépend de plein de facteurs : machine utilisée, langage, puissance, gestion mémoire...

- on compte le nb d'opérations élémentaires effectuées par l'ordinateur et la taille des objets utilisés
- préciser les opérations élémentaires en début d'analyse

Modèle retenu

Pour avoir l'ordre de grandeur, on s'intéresse au comportement asymptotique de la complexité **exprimée comme fonction de la taille de l'entrée**

Taille de l'entrée ?

- entier N représenté en base $b \rightarrow \lceil \log_b N \rceil$
- entier mod $n \rightarrow \log n$ (selon base)
- rationnel $\rightarrow$ somme de la taille du numérateur et dénominateur dans sa représentation
- polynôme de degré $d \rightarrow$ somme de la taille des coefficients dans l'anneau A
  - si $A = \mathbb{Z}/n\mathbb{Z} \rightarrow (d+1)\log n$
  - si $A = \mathbb{Q}, \mathbb{Z} \rightarrow$ il faut connaître une borne sur la taille des coefficients
- élément de $\mathbb{F}_{p^n}$ représenté par une classe de $\mathbb{F}_p[X]/(P)$ avec $P \in \mathbb{F}_p[X]$ irréductible de deg n
  $\rightarrow n \log p$
- un réel : approximation flottante $1.b_1 b_2 \ldots b_n \times 2^e$ avec e exposant entier borné, n taille de la mantisse fixée
  $\rightarrow$ taille constante (32 ou 64 bits selon la norme)

Complexité pire cas, meilleur cas, moyenne

Exemple

Recherche de la position d'un élément dans une liste non triée de N éléments distincts

Algorithme

Parcourir les éléments de la liste jusqu'à trouver le bon

Complexité pire cas, meilleur cas, moyenne

Exemple

Recherche de la position d'un élément dans une liste non triée de N éléments distincts

Algorithme

Parcourir les éléments de la liste jusqu'à trouver le bon

- meilleur cas : 1 comparaison
- pire cas : N comparaisons
- en moyenne : en supposant tous les ordres de la liste équiprobables, espérance du nombre de comparaisons nécessaires est $(N+1)/2$

$O(1)$ dans le meilleur cas, $O(N)$ sinon

Complexité pire cas, meilleur cas, moyenne

Ce qu'il faut retenir

- pour une complexité en moyenne, on doit fixer une loi de proba sur l'ensemble des entrées possibles de taille N chacune
- on s'intéresse à la complexité en moyenne ou au pire des cas

Ordres de grandeur

Que peut faire un ordinateur ?

- processeurs les plus rapides sont cadencés en GHz : 10^9 op/sec soit 2^{30} op/sec
- $10^6 (\simeq 2^{20})$ proc. pendant 30 ans ($\simeq 2^{30}$ sec) $\rightarrow$ borne sup en 2^{80} instructions

Ordres de grandeur

Que peut faire un ordinateur ?

- processeurs les plus rapides sont cadencés en GHz : 10^9 op/sec soit 2^{30} op/sec
- $10^6 (\simeq 2^{20})$ proc. pendant 30 ans ($\simeq 2^{30}$ sec) $\rightarrow$ borne sup en 2^{80} instructions

n	temps d'exécution avec 2^{30} op.s^{-1}				
	$\log_2(n)$	n	n^3	2^n	$n!$
10	3 ns	9 ns	0.9 μs	0.9 μs	3 ms
20	4 ns	18 ns	7 μs	1 ms	70 ans
30	4.5 ns	28 ns	25 μs	1 s	> âge univers
50	5.2 ns	46 ns	0.1 ms	12 jours	–
75	5.8 ns	70 ns	0.4 ms	1 millions d'années	–
100	6.2 ns	93 ns	0.9 ms	> âge univers	–
1000	9.2 ns	0.9 μs	1 s	–	–
10000	12 ns	9 μs	1000 s	–	–

Ordres de grandeur

En mémoire ?

- quantité d'information numérique sur terre en $10^{21} \simeq 2^{70}$ octets
- plus raisonnablement un algorithme avec plus de 2^{50} octets en mémoire est irréaliste

Ordres de grandeur

En mémoire ?

- quantité d'information numérique sur terre en $10^{21} \simeq 2^{70}$ octets
- plus raisonnablement un algorithme avec plus de 2^{50} octets en mémoire est irréaliste

Remarque

complexité en temps $>>$ complexité en mémoire

Section 2

Complexité des opérations arithmétiques usuelles

Opérations sur nombres flottants

Un nombre à virgule flottante est toujours stocké sur 32 ou 64 bits (**mot machine**).

Architecture matérielle dédiée pour les opérations sur ces nombres (l'unité arithmétique et logique, ALU)
$\rightsquigarrow$ complexité en $O(1)$ **(temps constant)**

Même chose pour les petits entiers ($< 2^{32}$ ou 2^{64}).

Opérations sur les entiers : addition

- Algorithme standard : on additionne deux entiers chiffre par chiffre, en allant de la droite vers la gauche, avec gestion des retenues
- Additionner deux (ou trois) chiffres est en **temps constant** : la table d'addition est connue !
- Même algorithme quelle que soit la **base utilisée** (2, 10, ou 2^{64})
- Même principe pour la **soustraction**

Opérations sur les entiers : addition

- Algorithme standard : on additionne deux entiers chiffre par chiffre, en allant de la droite vers la gauche, avec gestion des retenues
- Additionner deux (ou trois) chiffres est en **temps constant** : la table d'addition est connue !
- Même algorithme quelle que soit la **base utilisée** (2, 10, ou 2^{64})
- Même principe pour la **soustraction**

Complexité proportionnelle à la plus grande taille des deux entiers a et b : $O(\max(\log(a), \log(b)))$

Opérations sur les entiers : addition

- Algorithme standard : on additionne deux entiers chiffre par chiffre, en allant de la droite vers la gauche, avec gestion des retenues
- Additionner deux (ou trois) chiffres est en **temps constant** : la table d'addition est connue !
- Même algorithme quelle que soit la **base utilisée** (2, 10, ou 2^{64})
- Même principe pour la **soustraction**

Complexité proportionnelle à la plus grande taille des deux entiers a et b : $O(\max(\log(a), \log(b)))$

Questions :

– Pourquoi ça ne dépend pas de la base utilisée ?

– Pourquoi on ne peut pas faire mieux ?

Opérations sur les entiers : multiplication standard

- Multiplier deux chiffres est en **temps constant** : la table de multiplication est connue !

- Multiplier un nombre a de n chiffres par un nombre à 1 chiffre est en $O(n) = O(\log(a))$
  - n multiplications à un chiffre
  - au plus n additions à un ou deux chiffres (retenues)

$$\begin{array}{ccccc} {\scriptstyle 2} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 1} & \\ & 7 & 5 & 8 & 4 \\ \times & & & & 3 \\ \hline 2 & 2 & 7 & 5 & 2 \end{array}$$

Le résultat a au plus $n + 1$ chiffres.

Opérations sur les entiers : multiplication standard

Algorithme standard pour multiplier un nombre a de n chiffres par un nombre b de m chiffres est en $O(nm) = O(\log(a)\log(b))$:

$$
\begin{array}{rrrrrrrr}
 & & & 7 & 5 & 8 & 4 \\
\times & & & & 6 & 5 & 3 \\
\hline
 & & 2 & 2 & 7 & 5 & 2 \\
 & 3 & 7 & 9 & 2 & 0 & 0 \\
4 & 5 & 5 & 0 & 4 & 0 & 0 \\
\hline
4 & 9 & 5 & 2 & 3 & 5 & 2
\end{array}
$$

- m multiplications de a par un chiffre
- additions de m nombres à au plus $m+n$ chiffres

Opérations sur les entiers : multiplication standard

Algorithme standard pour multiplier un nombre a de n chiffres par un nombre b de m chiffres est en $O(nm) = O(\log(a)\log(b))$:

			7	5	8	4
×				6	5	3
		2	2	7	5	2
	3	7	9	2	0	0
4	5	5	0	4	0	0
4	9	5	2	3	5	2

- m multiplications de a par un chiffre
- additions de m nombres à au plus $m + n$ chiffres

Même algorithme quelle que soit la **base utilisée** (2, 10, ou 2^{64})

Complexité proportionnelle au produit des tailles des deux entiers a et b : $O(\log(a)\log(b))$

Opérations sur les entiers : multiplication rapide

Pour deux entiers de taille n : produit standard en $O(n^2)$ (quadratique)

On peut faire mieux !

- Algorithme de Karatsuba (diviser pour régner) : $O(n^{\log_2(3)}) = O(n^{1.549\ldots})$
- Algorithme de Harvey–van der Hoeven (transformée de Fourier rapide, en 2019 !) : $O(n\log(n))$

Opérations sur les entiers : multiplication rapide

Pour deux entiers de taille n : produit standard en $O(n^2)$ (quadratique)

On peut faire mieux !

- Algorithme de Karatsuba (diviser pour régner) : $O(n^{\log_2(3)}) = O(n^{1.549\ldots})$
- Algorithme de Harvey–van der Hoeven (transformée de Fourier rapide, en 2019 !) : $O(n \log(n))$

Attention aux constantes cachées ! Les complexités sont *asymptotiques*.
$\longrightarrow$ diviser-pour-régner intéressant à partir d'une centaine de chiffres
$\longrightarrow$ Fourier intéressant à partir de plusieurs milliers de chiffres

Opérations sur les entiers : division euclidienne

Algorithme standard : traite le dividende chiffre par chiffre

	7	9	8	4	1	3	7	1
−	7	4	2			2	1	5
		5	6	4				
	−	3	7	1				
		1	9	3	1			
	−	1	8	5	5			
				7	6			

Opérations sur les entiers : division euclidienne

Algorithme standard : traite le dividende chiffre par chiffre

Pour diviser a de taille n par b de taille m :

- $n - m + 1$ étapes
- à chaque étape :
  – produit de b par un chiffre, répété $O(1)$ fois (pour trouver le bon chiffre du quotient)
  – une soustraction d'entiers de taille m ou $m + 1$

Opérations sur les entiers : division euclidienne

Algorithme standard : traite le dividende chiffre par chiffre

Pour diviser a de taille n par b de taille m :

- $n - m + 1$ étapes
- à chaque étape :
  – produit de b par un chiffre, répété $O(1)$ fois (pour trouver le bon chiffre du quotient)
  – une soustraction d'entiers de taille m ou $m + 1$

Complexité en $O((n - m + 1)m)$

Au pire (cas $m \approx n/2$), en $O(n^2)$: complexité quadratique.

Algorithmes rapides : division euclidienne en $O(n \log(n))$.

Opérations sur les entiers : division euclidienne

Remarque :

Pour b proche d'une puissance de la base utilisée, la division / réduction modulaire par b est beaucoup plus rapide

Exemples :

$20\,212\,022 = 2\,022 + 2\,021 \times 10\,000$

$20\,212\,022 = \qquad [10\,000]$

$20\,212\,022 = \qquad [10\,001]$

$20\,212\,022 = \qquad [9\,980]$

à comparer à :

$20\,212\,022 = \qquad [6\,183]$

Opérations sur les entiers : division euclidienne

Remarque :
Pour b proche d'une puissance de la base utilisée, la division / réduction modulaire par b est beaucoup plus rapide

Exemples :

$$20\,212\,022 = 2\,022 + 2\,021 \times 10\,000$$
$$20\,212\,022 = 2\,022 \ [10\,000]$$
$$20\,212\,022 = \qquad [10\,001]$$
$$20\,212\,022 = \qquad [9\,980]$$

à comparer à :

$$20\,212\,022 = \qquad [6\,183]$$

Opérations sur les entiers : division euclidienne

Remarque :

Pour b proche d'une puissance de la base utilisée, la division / réduction modulaire par b est beaucoup plus rapide

Exemples :

$20\,212\,022 = 2\,022 + 2\,021 \times 10\,000$

$20\,212\,022 = 2\,022\ [10\,000]$

$20\,212\,022 = 2\,022 - 2\,021 = 1\ [10\,001]$

$20\,212\,022 = \qquad [9\,980]$

à comparer à :

$20\,212\,022 = \qquad [6\,183]$

Opérations sur les entiers : division euclidienne

Remarque :
Pour b proche d'une puissance de la base utilisée, la division / réduction modulaire par b est beaucoup plus rapide

Exemples :
$20\,212\,022 = 2\,022 + 2\,021 \times 10\,000$
$20\,212\,022 = 2\,022\ [10\,000]$
$20\,212\,022 = 2\,022 - 2\,021 = 1\ [10\,001]$
$20\,212\,022 = 2\,022 + 20 \times 2\,021 = 42\,442 = 2\,442 + 20 \times 4 = 2\,522\ [9\,980]$

à comparer à :
$20\,212\,022 = \qquad [6\,183]$

Opérations sur les entiers : division euclidienne

Remarque :

Pour b proche d'une puissance de la base utilisée, la division / réduction modulaire par b est beaucoup plus rapide

Exemples :

$20\,212\,022 = 2\,022 + 2\,021 \times 10\,000$

$20\,212\,022 = 2\,022 \; [10\,000]$

$20\,212\,022 = 2\,022 - 2\,021 = 1 \; [10\,001]$

$20\,212\,022 = 2\,022 + 20 \times 2\,021 = 42\,442 = 2\,442 + 20 \times 4 = 2\,522 \; [9\,980]$

à comparer à :

$20\,212\,022 = 5\,978 \; [6\,183]$

Opérations sur les entiers : pgcd et Euclide étendu

Rappel : on suppose a, b positifs.
On définit par récurrence la suite (r_i) par :

$r_0 = a$
$r_1 = b$
Tant que $r_i \neq 0$:
  $q_i =$ quotient division de r_{i-1} par r_i
  $r_{i+1} = r_{i-1} - q_i r_i$ (reste division de r_{i-1} par r_i)

$a \wedge b =$ dernier reste non nul

Invariant de boucle : $r_i \wedge r_{i+1} = a \wedge b$

Opérations sur les entiers : pgcd et Euclide étendu

Rappel : on suppose a, b positifs.
On définit par récurrence la suite (r_i) et les suites (u_i), (v_i) par :

$$
\begin{array}{lll}
r_0 = a & u_0 = 1 & v_0 = 0 \\
r_1 = b & u_1 = 0 & v_1 = 1
\end{array}
$$

Tant que $r_i \neq 0$:

$q_i =$ quotient division de r_{i-1} par r_i

$r_{i+1} = r_{i-1} - q_i r_i$ (reste division de r_{i-1} par r_i)

$u_{i+1} = u_{i-1} - q_i u_i$

$v_{i+1} = v_{i-1} - q_i v_i$

$a \wedge b =$ dernier reste non nul

Invariant de boucle : $r_i \wedge r_{i+1} = a \wedge b \qquad au_i + bv_i = r_i$

Opérations sur les entiers : pgcd et Euclide étendu

Rappel : on suppose a, b positifs.
On définit par récurrence la suite (r_i) et les suites (u_i), (v_i) par :

$r_0 = a \qquad u_0 = 1 \qquad v_0 = 0$
$r_1 = b \qquad u_1 = 0 \qquad v_1 = 1$
Tant que $r_i \neq 0$:
$\quad q_i =$ quotient division de r_{i-1} par r_i
$\quad r_{i+1} = r_{i-1} - q_i r_i$ (reste division de r_{i-1} par r_i)
$\quad u_{i+1} = u_{i-1} - q_i u_i$
$\quad v_{i+1} = v_{i-1} - q_i v_i$

$a \wedge b =$ dernier reste non nul

Invariant de boucle : $\quad r_i \wedge r_{i+1} = a \wedge b \qquad au_i + bv_i = r_i$

Au dernier indice on a bien $au_i + bv_i = a \wedge b$ (**coefficients de Bézout**)

Opérations sur les entiers : pgcd et Euclide étendu

Complexité : on suppose $b \leqslant a$

- Division euclidienne de r_{i-1} par r_i en $O((\log r_{i-1} - \log r_i) \log r_i)$
  $\rightsquigarrow$ on majore par $O((\log r_{i-1} - \log r_i) \log a)$.
- Télescopage : complexité globale en $O((\log a - \log(a \wedge b)) \log(a))$
- Analyse similaire pour les coefficients de Bézout

Pour a, b de taille majorée par n, complexité d'Euclide étendu en $O(n^2)$ (quadratique)

Opérations sur les entiers : pgcd et Euclide étendu

Complexité : on suppose $b \leqslant a$

- Division euclidienne de r_{i-1} par r_i en $O((\log r_{i-1} - \log r_i)\log r_i)$
  $\rightsquigarrow$ on majore par $O((\log r_{i-1} - \log r_i)\log a)$.
- Télescopage : complexité globale en $O((\log a - \log(a \wedge b))\log(a))$
- Analyse similaire pour les coefficients de Bézout

Pour a, b de taille majorée par n, complexité d'Euclide étendu en $O(n^2)$ (quadratique)

Algorithmes rapides : en $O(n \log(n)^2)$

Arithmétique modulaire : opérations de base

Dans $\mathbb{Z}/N\mathbb{Z}$: éléments a, b donnés par leur représentant dans $[\![0, N-1]\!]$

- **Addition :** $a+b$ dans $\mathbb{Z}$ en $O(\log(N))$, résultat dans $[\![0, 2N-1]\!]$.
  On retranche N en $O(\log(N))$ si la somme dépasse
  $\rightsquigarrow$ total en $O(\log(N))$

Arithmétique modulaire : opérations de base

Dans $\mathbb{Z}/N\mathbb{Z}$: éléments a, b donnés par leur représentant dans $[\![0, N-1]\!]$

- **Addition :** $a + b$ dans $\mathbb{Z}$ en $O(\log(N))$, résultat dans $[\![0, 2N-1]\!]$.
  On retranche N en $O(\log(N))$ si la somme dépasse
  $\rightsquigarrow$ total en $O(\log(N))$
- **Produit :** $a \times b$ dans $\mathbb{Z}$ en $O(\log(N)^2)$, résultat dans $[\![0, (N-1)^2]\!]$.
  Division euclidienne par N pour le représentant, en $O(\log(N)^2)$
  $\rightsquigarrow$ total en $O(\log(N)^2)$
- **Inversion :** Euclide étendu entre a et N $\rightsquigarrow$ total en $O(\log(N)^2)$

Arithmétique modulaire : opérations de base

Dans $\mathbb{Z}/N\mathbb{Z}$: éléments a, b donnés par leur représentant dans $[\![0, N-1]\!]$

- **Addition :** $a+b$ dans $\mathbb{Z}$ en $O(\log(N))$, résultat dans $[\![0, 2N-1]\!]$.
  On retranche N en $O(\log(N))$ si la somme dépasse
  $\rightsquigarrow$ total en $O(\log(N))$
- **Produit :** $a \times b$ dans $\mathbb{Z}$ en $O(\log(N)^2)$, résultat dans $[\![0, (N-1)^2]\!]$.
  Division euclidienne par N pour le représentant, en $O(\log(N)^2)$
  $\rightsquigarrow$ total en $O(\log(N)^2)$
- **Inversion :** Euclide étendu entre a et N $\rightsquigarrow$ total en $O(\log(N)^2)$

En pratique

– Additions dans $\mathbb{Z}/N\mathbb{Z}$ souvent négligeables par rapport au reste
– Inversions sensiblement plus lentes que produits
– Mise au carré un peu plus rapide que produit quelconque

Arithmétique modulaire : exponentiation rapide

Exponentiation modulaire : a donné par son représentant dans $[\![0, N-1]\!]$, $n \in \mathbb{N}$, trouver un représentant de a^n dans $[\![0, N-1]\!]$

Algorithme

$$a^n = \prod_{i, \epsilon_i = 1} a^{2^i} \text{ avec } n = \sum_i \epsilon_i 2^i \text{ écriture binaire de } n$$

→ mises au carré successives de a et multiplications selon les bits de n

```
def fast_exp_right_to_left(a,n):
    res=1
    t=a
    while n>0:
        if n%2==1: res*=t
        n=n//2
        t=t^2
    return res
```

Arithmétique modulaire : exponentiation rapide

Algorithme version gauche à droite

$$a^n = \begin{cases} (a^{n/2})^2 \\ a(a^{\frac{n-1}{2}})^2 \end{cases}$$

- version récursive → version itérative :
  lecture des bits de n de poids fort vers poids faible
- mises au carré successives de a et multiplications selon les bits de n

```
def fast_exp_left_to_right(a,n):
    res=1
    L=n.bits()
    L.reverse()
    for x in L:
        res=res^2
        if x==1: res*=a
    return res
```

Arithmétique modulaire : exponentiation rapide

Principe : mises au carré successives de $a \in \mathbb{Z}/N\mathbb{Z}$ et multiplications selon les bits de n

$$a^n = \prod_{i, \epsilon_i = 1} a^{2^i} \text{ avec } n = (\epsilon_b \ldots \epsilon_0) \text{ où } b = \lfloor \log_2 n \rfloor$$

Complexité

- b mises au carrés dans $\mathbb{Z}/N\mathbb{Z} \rightarrow O(b(\log N)^2)$
- au plus b multiplications dans $\mathbb{Z}/N\mathbb{Z} \rightarrow O(b(\log N)^2)$
- Euler-Fermat $\rightarrow$ en général $n < N$

Complexité totale en $O((\log N)^3)$

Arithmétique modulaire : exponentiation rapide

Principe : mises au carré successives de $a \in \mathbb{Z}/N\mathbb{Z}$ et multiplications selon les bits de n

$$a^n = \prod_{i,\epsilon_i=1} a^{2^i} \text{ avec } n = (\epsilon_b \dots \epsilon_0) \text{ où } b = \lfloor \log_2 n \rfloor$$

Complexité

- b mises au carrés dans $\mathbb{Z}/N\mathbb{Z} \to O(b(\log N)^2)$
- au plus b multiplications dans $\mathbb{Z}/N\mathbb{Z} \to O(b(\log N)^2)$
- Euler-Fermat $\to$ en général $n < N$

Complexité totale en $O((\log N)^3)$

Remarque : calcul possible d'inverse modulaire par

$$a^{-1} = a^{\varphi(N)-1} \in \mathbb{Z}/N\mathbb{Z}^\times$$

mais moins efficace qu'Euclide étendu...

Arithmétique des polynômes

A = anneau commutatif unitaire intègre

Représentation pleine ou creuse

Un élément de $A[X]$ peut être représenté

- par la liste / tableau de ses coefficients : **représentation pleine**
  $X^9 + 2X^5 - X^2 - 3X + 1 \longleftrightarrow [1, -3, -1, 0, 0, 2, 0, 0, 0, 1]$
  (attention aux zéros qui trainent)
- par une liste de couples degré-coefficient : **représentation creuse**
  $X^9 + 2X^5 - X^2 - 3X + 1 \longleftrightarrow [(0, 1), (1, -3), (2, -1), (5, 2), (9, 1)]$

On se limitera à la représentation pleine.
Taille de $P \in A[X]$ = taille de $\deg(P) + 1$ éléments de A.

Arithmétique des polynômes : opérations de base

Algorithmes proches de ceux dans $\mathbb{Z}$, **sans** gestion de retenue.
Pour $P, Q \in A[X]$, de degrés $< D$:

- **Addition/soustraction :** $P + Q$ en $\min(\deg(P) + 1, \deg(Q) + 1))$ additions/soustractions dans A et recopie des coefficients jusqu'à $\max(\deg(P), \deg(Q))$
  $\rightsquigarrow O(D)$ additions dans A
- **Multiplication :** $PQ = \sum_{i=0}^{\deg(P)} \sum_{j=0}^{\deg(Q)} p_i q_j X^{i+j}$,
  d'où $(\deg(P) + 1)(\deg(Q) + 1)$ produits et additions dans A
  $\rightsquigarrow O(D^2)$ multiplications dans A (on néglige les additions)

Arithmétique des polynômes : opérations de base

Algorithmes proches de ceux dans $\mathbb{Z}$, **sans** gestion de retenue.
Pour $P, Q \in A[X]$, de degrés $< D$:

- **Addition/soustraction :** $P + Q$ en $\min(\deg(P) + 1, \deg(Q) + 1))$ additions/soustractions dans A et recopie des coefficients jusqu'à $\max(\deg(P), \deg(Q))$
  $\rightsquigarrow O(D)$ additions dans A
- **Multiplication :** $PQ = \sum_{i=0}^{\deg(P)} \sum_{j=0}^{\deg(Q)} p_i q_j X^{i+j}$,
  d'où $(\deg(P) + 1)(\deg(Q) + 1)$ produits et additions dans A
  $\rightsquigarrow O(D^2)$ multiplications dans A (on néglige les additions)

Attention !

Pertinent si on peut majorer le coût des opérations dans A (flottants, $\mathbb{Z}/N\mathbb{Z}$,...)
Si $A = \mathbb{Z}$ ou $\mathbb{Q}$: besoin aussi d'une majoration sur les **coefficients**

Arithmétique des polynômes : opérations (suite)

- **Division euclidienne :** le coefficient dominant de Q doit être *inversible*
  $\deg(P) - \deg(Q) + 1$ étapes, à chaque fois $\deg(Q) + 1$ produits et soustractions dans A
  $\rightsquigarrow$ en $O((\deg(P) - \deg(Q) + 1)\deg(Q))$ produits dans A (plus une inversion)

- **Euclide étendu :** même analyse que dans $\mathbb{Z}$
  - $O(D^2)$ produits
  - $O(D)$ inversions dans A (1 inversion à chaque division euclidienne, et au plus D divisions euclidiennes)

Arithmétique des polynômes : opérations (suite)

- **Division euclidienne :** le coefficient dominant de Q doit être *inversible*
  $\deg(P) - \deg(Q) + 1$ étapes, à chaque fois $\deg(Q) + 1$ produits et soustractions dans A
  $\rightsquigarrow$ en $O((\deg(P) - \deg(Q) + 1)\deg(Q))$ produits dans A (plus une inversion)

- **Euclide étendu :** même analyse que dans $\mathbb{Z}$
  - $O(D^2)$ produits
  - $O(D)$ inversions dans A (1 inversion à chaque division euclidienne, et au plus D divisions euclidiennes)

Algorithmes rapides : en utilisant diviser-pour-régner ou Fourier, on peut descendre jusqu'à $O(D\log(D))$ pour produit et division et $O(D\log(D)^2)$ pour Euclide, au moins sur les corps finis.

Arithmétique des polynômes : évaluation

But : calculer $P(a) = \sum_{k=0}^{n} c_k a^k$.

Évidemment exclus de repartir de zéro pour calculer x^k à chaque fois...

Méthode naïve : deux produits à chaque étape (passage $a^{k-1} \to a^k$ puis calcul $c_k a^k$) et une addition, sauf pour $k \leqslant 1$
$\rightsquigarrow$ au total $2n - 1$ produits et n additions dans A

Arithmétique des polynômes : évaluation

But : calculer $P(a) = \sum_{k=0}^{n} c_k a^k$.

Évidemment exclus de repartir de zéro pour calculer x^k à chaque fois...

Méthode naïve : deux produits à chaque étape (passage $a^{k-1} \to a^k$ puis calcul $c_k a^k$) et une addition, sauf pour $k \leqslant 1$
$\rightsquigarrow$ au total $2n-1$ produits et n additions dans A

On peut faire (un peu) mieux :

Schéma de Horner

Expression $P(a) = ((\dots((c_n a + c_{n-1})a + c_{n-2})\,a + \dots)\,a + c_1)\,a + c_0$

Permet l'évaluation en n produits et n additions dans A

Arithmétique des polynômes : interpolation

But : étant donnés $x_1, \ldots, x_D \in A$ tels que $x_i - x_j \in A^\times$ et $y_1, \ldots, y_D \in A$ calculer

$$P = \sum_{i=1}^{D} y_i \; \frac{\prod_{j \neq i}(X - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

l'unique polynôme $P \in A[X]_{<D}$ tel que $P(x_i) = y_i$

Arithmétique des polynômes : interpolation

But : étant donnés $x_1, \ldots, x_D \in A$ tels que $x_i - x_j \in A^\times$ et $y_1, \ldots, y_D \in A$ calculer

$$P = \sum_{i=1}^{D} y_i \, \frac{\prod_{j \neq i}(X - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

l'unique polynôme $P \in A[X]_{<D}$ tel que $P(x_i) = y_i$

Méthode

- calculer $L = \prod_j (X - x_j)$ de proche en proche $\rightarrow O(D^2)$ opérations dans A

Arithmétique des polynômes : interpolation

But : étant donnés $x_1, \ldots, x_D \in A$ tels que $x_i - x_j \in A^\times$ et $y_1, \ldots, y_D \in A$ calculer

$$P = \sum_{i=1}^{D} y_i \frac{\prod_{j \neq i}(X - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

l'unique polynôme $P \in A[X]_{<D}$ tel que $P(x_i) = y_i$

Méthode

- calculer $L = \prod_j (X - x_j)$ de proche en proche $\rightarrow O(D^2)$ opérations dans A
- calculer $L_i = \prod_{j \neq i}(X - x_j)$ par la division (exacte) de L par $X - x_i$ en $O(D)$

Arithmétique des polynômes : interpolation

But : étant donnés $x_1, \ldots, x_D \in A$ tels que $x_i - x_j \in A^\times$ et $y_1, \ldots, y_D \in A$ calculer

$$P = \sum_{i=1}^{D} y_i \; \frac{\prod_{j \neq i}(X - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

l'unique polynôme $P \in A[X]_{<D}$ tel que $P(x_i) = y_i$

Méthode

- calculer $L = \prod_j (X - x_j)$ de proche en proche $\rightarrow O(D^2)$ opérations dans A
- calculer $L_i = \prod_{j \neq i}(X - x_j)$ par la division (exacte) de L par $X - x_i$ en $O(D)$
- évaluer L_i en x_i en $O(D) \rightsquigarrow P = \sum_{i=1}^{D} y_i \frac{L_i(X)}{L_i(x_i)}$ en $O(D^2)$

Arithmétique des polynômes : interpolation

But : étant donnés $x_1, \ldots, x_D \in A$ tels que $x_i - x_j \in A^\times$ et $y_1, \ldots, y_D \in A$ calculer

$$P = \sum_{i=1}^{D} y_i \, \frac{\prod_{j \neq i}(X - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

l'unique polynôme $P \in A[X]_{<D}$ tel que $P(x_i) = y_i$

Méthode

- calculer $L = \prod_j (X - x_j)$ de proche en proche $\rightarrow O(D^2)$ opérations dans A
- calculer $L_i = \prod_{j \neq i}(X - x_j)$ par la division (exacte) de L par $X - x_i$ en $O(D)$
- évaluer L_i en x_i en $O(D) \rightsquigarrow P = \sum_{i=1}^{D} y_i \frac{L_i(X)}{L_i(x_i)}$ en $O(D^2)$

Complexité totale $O(D^2)$ opérations dans A

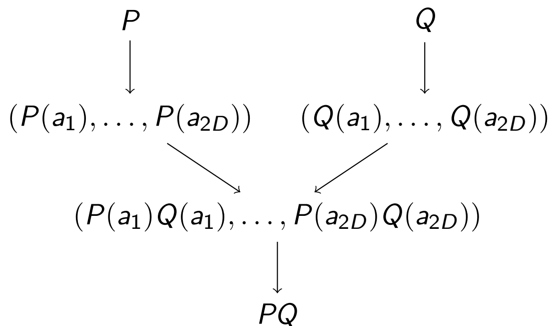
Section 3

Algorithmes rapides sur les polynômes

Multiplication rapide de polynômes dans $\mathbb{C}$: principe

Soit P, Q de degré $< D$.

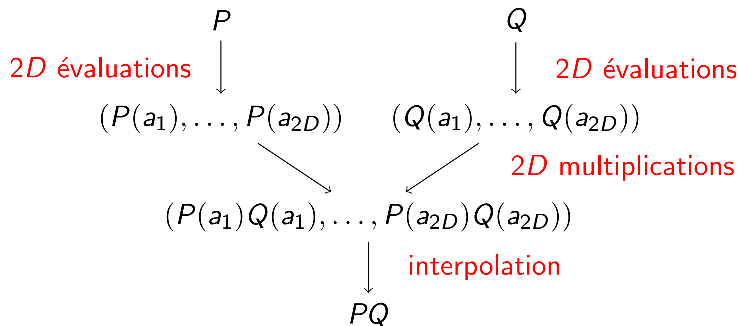
Idée : PQ déterminé par ses valeurs en $2D$ points (*Lagrange*)



Multiplication rapide de polynômes dans $\mathbb{C}$: principe

Soit P, Q de degré $< D$.

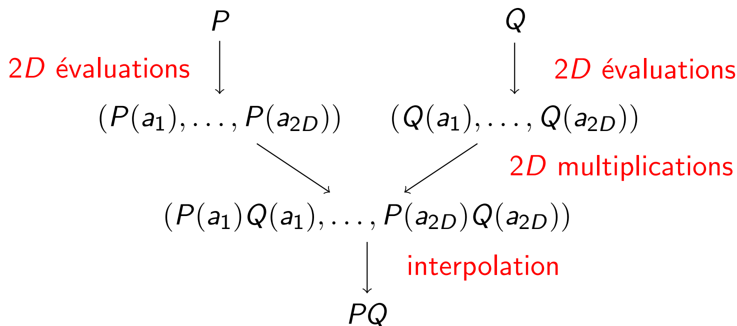
Idée : PQ déterminé par ses valeurs en $2D$ points (*Lagrange*)



Multiplication rapide de polynômes dans $\mathbb{C}$: principe

Soit P, Q de degré $< D$.

Idée : PQ déterminé par ses valeurs en $2D$ points (*Lagrange*)



Évaluation naïve : $O(D)$ produits $\rightsquigarrow$ total en $O(D^2)$, pas pertinent.
Intéressant si évaluation et interpolation rapides

Multiplication rapide de polynômes dans $\mathbb{C}$: principe

Idée 2 : évaluation en les $2D$ racines de l'unité

Si $P = \sum_{k=0}^{D-1} p_k X^k$, alors pour tout $\ell \in [\![0, 2D-1]\!]$,

$$P(e^{-\frac{2i\pi}{2D}\ell}) = \sum_{k=0}^{D-1} p_k \, e^{-\frac{2i\pi}{2D}\ell k}$$

Multiplication rapide de polynômes dans $\mathbb{C}$: principe

Idée 2 : évaluation en les $2D$ racines de l'unité

Si $P = \sum_{k=0}^{D-1} p_k X^k$, alors pour tout $\ell \in [\![0, 2D-1]\!]$,

$$P(e^{-\frac{2i\pi}{2D}\ell}) = \sum_{k=0}^{D-1} p_k \, e^{-\frac{2i\pi}{2D}\ell k}$$

On reconnaît la **transformée de Fourier discrète** du $2D$-uplet $(p_0, \ldots, p_{D-1}, 0, \ldots, 0)$.

$\rightsquigarrow$ reconstruction du produit PQ à partir de $\left(PQ(e^{-\frac{2i\pi}{2D}\ell})\right)_{0 \leqslant \ell < 2D}$ par **transformée de Fourier discrète inverse** (et pas Lagrange !).

Rappels sur la transformée de Fourier discrète

On note $\omega_N = e^{\frac{2i\pi}{N}}$, d'où $\mathbb{U}_N = \{\omega_N^\ell \mid \ell \in [\![0, N-1]\!]\}$

Transformée de Fourier discrète

Passage entre deux bases de $\mathbb{C}^N$

- **base canonique** $\mathcal{B}_c = (e_0, \ldots, e_{N-1})$ où $e_\ell = (0, \ldots, 0, 1, 0, \ldots, 0) = (\delta_{\ell k})_{0 \leqslant k < N-1}$
- **base "spectrale"** $\mathcal{B}_s = (v_0, \ldots, v_{N-1})$ où $v_\ell = (1, \omega_N^\ell, \omega_N^{2\ell}, \ldots, \omega_N^{(N-1)\ell}) = (\omega_N^{\ell k})_{0 \leqslant k < N-1}$

Rappels sur la transformée de Fourier discrète

On note $\omega_N = e^{\frac{2i\pi}{N}}$, d'où $\mathbb{U}_N = \{\omega_N^\ell \mid \ell \in [\![0, N-1]\!]\}$

Transformée de Fourier discrète

Passage entre deux bases de $\mathbb{C}^N$

- **base canonique** $\mathcal{B}_c = (e_0, \ldots, e_{N-1})$ où
  $e_\ell = (0, \ldots, 0, 1, 0, \ldots, 0) = (\delta_{\ell k})_{0 \leqslant k < N-1}$
- **base "spectrale"** $\mathcal{B}_s = (v_0, \ldots, v_{N-1})$ où
  $v_\ell = (1, \omega_N^\ell, \omega_N^{2\ell}, \ldots, \omega_N^{(N-1)\ell}) = (\omega_N^{\ell k})_{0 \leqslant k < N-1}$

À un facteur $\sqrt{N}$ près, les deux bases sont orthonormées
$\rightsquigarrow$ transformation *unitaire* de $\mathbb{C}^N$.

Rappels sur la transformée de Fourier discrète

Soit $u = (u_0, \ldots, u_{N-1}) = (u_k)_{0 \leqslant k < N} \in \mathbb{C}^N$

Formules

- Transformée de Fourier directe :

$$\hat{u} = (\hat{u}_0, \ldots, \hat{u}_{N-1}) = (\hat{u}_\ell)_{0 \leqslant \ell < N} \in \mathbb{C}^N \text{ où } \hat{u}_\ell = \sum_{k=0}^{N-1} u_k \, \omega_N^{-\ell k}$$

- Transformée de Fourier inverse : pour tout k,

$$u_k = \frac{1}{N} \sum_{\ell=0}^{N-1} \hat{u}_\ell \, \omega_N^{\ell k}$$

Rappels sur la transformée de Fourier discrète

Soit $u = (u_0, \dots, u_{N-1}) = (u_k)_{0 \leqslant k < N} \in \mathbb{C}^N$

Formules

- Transformée de Fourier directe :

$$\hat{u} = (\hat{u}_0, \dots, \hat{u}_{N-1}) = (\hat{u}_\ell)_{0 \leqslant \ell < N} \in \mathbb{C}^N \text{ où } \hat{u}_\ell = \sum_{k=0}^{N-1} u_k\, \omega_N^{-\ell k}$$

- Transformée de Fourier inverse : pour tout k,

$$u_k = \frac{1}{N} \sum_{\ell=0}^{N-1} \hat{u}_\ell\, \omega_N^{\ell k}$$

Remarques

- place du $1/N$ varie selon les auteurs

Rappels sur la transformée de Fourier discrète

Soit $u = (u_0, \ldots, u_{N-1}) = (u_k)_{0 \leqslant k < N} \in \mathbb{C}^N$

Formules

- Transformée de Fourier directe :

$$\hat{u} = (\hat{u}_0, \ldots, \hat{u}_{N-1}) = (\hat{u}_\ell)_{0 \leqslant \ell < N} \in \mathbb{C}^N \text{ où } \hat{u}_\ell = \frac{1}{N} \sum_{k=0}^{N-1} u_k \, \omega_N^{-\ell k}$$

- Transformée de Fourier inverse : pour tout k,

$$u_k = \sum_{\ell=0}^{N-1} \hat{u}_\ell \, \omega_N^{\ell k}$$

Remarques

- place du $1/N$ varie selon les auteurs

Rappels sur la transformée de Fourier discrète

Soit $u = (u_0, \ldots, u_{N-1}) = (u_k)_{0 \leqslant k < N} \in \mathbb{C}^N$

Formules

- Transformée de Fourier directe :

$$\hat{u} = (\hat{u}_0, \ldots, \hat{u}_{N-1}) = (\hat{u}_\ell)_{0 \leqslant \ell < N} \in \mathbb{C}^N \text{ où } \hat{u}_\ell = \frac{1}{N} \sum_{k=0}^{N-1} u_k \, \omega_N^{-\ell k}$$

- Transformée de Fourier inverse : pour tout k,

$$u_k = \sum_{\ell=0}^{N-1} \hat{u}_\ell \, \omega_N^{\ell k}$$

Remarques

- place du $1/N$ varie selon les auteurs
- généralisation à tout corps K contenant une racine primitive N-ième de l'unité ω_N (impose $\mathrm{char}(K) \nmid N$)

Transformée de Fourier discrète et convolution

Deux lois d'algèbre sur $\mathbb{C}^N$:

1. **Produit terme à terme** : $(u.v)_k = u_k\, v_k$
   $\rightsquigarrow$ structure d'anneau produit sur $\mathbb{C}^N$
2. **Produit de convolution** cyclique : $(u * v)_k = \sum_{j=0}^{N-1} u_j\, v_{k-j}$

   (avec indices pris modulo N)

Transformée de Fourier discrète et convolution

Deux lois d'algèbre sur $\mathbb{C}^N$:

1. **Produit terme à terme** : $(u.v)_k = u_k\, v_k$
   $\rightsquigarrow$ structure d'anneau produit sur $\mathbb{C}^N$
2. **Produit de convolution** cyclique : $(u * v)_k = \sum_{j=0}^{N-1} u_j\, v_{k-j}$

   (avec indices pris modulo N)

Convolution et produit de polynômes

Si $R = PQ$, alors $R = \sum_{k \geqslant 0} (\sum_{j=0}^{k} p_j\, q_{k-j}) X^k$.

En **complétant** par suffisamment de zéros, c'est une convolution cyclique :

$$(r_0, \ldots, r_{\deg(P)+\deg(Q)}, 0, \ldots, 0) = (p_0, \ldots, p_{\deg(P)}, 0, \ldots, 0) * (q_0, \ldots, q_{\deg(Q)}, 0, \ldots, 0)$$

Transformée de Fourier discrète et convolution

Propriété

La transformée de Fourier discrète $\mathcal{F}$ **échange** les deux lois d'algèbre :

$$\mathcal{F}(u * v) = \mathcal{F}(u).\mathcal{F}(v)$$

Pour le produit $R = PQ$, avec $N \geqslant \deg(P) + \deg(Q)$:

$$(r_0, \ldots, r_{\deg(P)+\deg(Q)}, 0, \ldots, 0)$$
$$= \mathcal{F}^{-1}\left(\mathcal{F}(p_0, \ldots, p_{\deg(P)}, 0, \ldots, 0).\mathcal{F}(q_0, \ldots, q_{\deg(Q)}, 0, \ldots, 0)\right)$$

Réinterprétation de l'idée de départ (évaluation + produits terme à terme + interpolation)

Transformée de Fourier rapide : Cooley-Tukey

Principe : **diviser pour régner**. On suppose N pair : $N = 2N'$

1. séparation de u en termes pairs et impairs :
   $v = (u_0, u_2, \ldots, u_{2N'-2})$ et $w = (u_1, u_3, \ldots, u_{2N'-1})$

Transformée de Fourier rapide : Cooley-Tukey

Principe : **diviser pour régner**. On suppose N pair : $N = 2N'$

1. séparation de u en termes pairs et impairs :
   $v = (u_0, u_2, \ldots, u_{2N'-2})$ et $w = (u_1, u_3, \ldots, u_{2N'-1})$
2. calcul des *deux transformées de taille moitié* :

$$\hat{v} = \left(\sum_{k=0}^{N'-1} u_{2k}\, \omega_{N'}^{-\ell k} \right)_{0 \leqslant \ell < N'} \quad \text{et} \quad \hat{w} = \left(\sum_{k=0}^{N'-1} u_{2k+1}\, \omega_{N'}^{-\ell k} \right)_{0 \leqslant \ell < N'}$$

Transformée de Fourier rapide : Cooley-Tukey

Principe : **diviser pour régner**. On suppose N pair : $N = 2N'$

1. séparation de u en termes pairs et impairs :
   $v = (u_0, u_2, \dots, u_{2N'-2})$ et $w = (u_1, u_3, \dots, u_{2N'-1})$
2. calcul des *deux transformées de taille moitié* :
   $$\hat{v} = \left(\sum_{k=0}^{N'-1} u_{2k}\, \omega_{N'}^{-\ell k} \right)_{0 \leqslant \ell < N'} \quad \text{et} \quad \hat{w} = \left(\sum_{k=0}^{N'-1} u_{2k+1}\, \omega_{N'}^{-\ell k} \right)_{0 \leqslant \ell < N'}$$
3. réunion : pour tout $\ell \in [\![0, 2N'-1]\!]$,
   $$\begin{aligned} \hat{u}_\ell &= \sum_{k=0}^{2N'-1} u_k\, \omega_N^{-\ell k} = \sum_{k=0}^{N'-1} u_{2k}\, \omega_N^{-2k\ell} + \sum_{k=0}^{N'-1} u_{2k+1}\, \omega_N^{-(2k+1)\ell} \\ &= \sum_{k=0}^{N'-1} u_{2k} (\omega_N^2)^{-k\ell} + \sum_{k=0}^{N'-1} u_{2k+1}\, \omega_N^{-\ell} (\omega_N^2)^{-k\ell} \\ &= \hat{v}_\ell + \omega_N^{-\ell} \hat{w}_\ell \qquad \text{puisque } \omega_N^2 = \omega_{N'} \end{aligned}$$

Transformée de Fourier rapide : Cooley-Tukey

Principe : **diviser pour régner**. On suppose N pair : $N = 2N'$

1. séparation de u en termes pairs et impairs :
   $v = (u_0, u_2, \ldots, u_{2N'-2})$ et $w = (u_1, u_3, \ldots, u_{2N'-1})$
2. calcul des *deux transformées de taille moitié* : à faire récursivement

$$\hat{v} = \left(\sum_{k=0}^{N'-1} u_{2k}\, \omega_{N'}^{-\ell k}\right)_{0\leqslant \ell < N'} \qquad \text{et} \qquad \hat{w} = \left(\sum_{k=0}^{N'-1} u_{2k+1}\, \omega_{N'}^{-\ell k}\right)_{0\leqslant \ell < N'}$$

3. réunion : pour tout $\ell \in [\![0, 2N'-1]\!]$,

$$\begin{aligned}
\hat{u}_\ell &= \sum_{k=0}^{2N'-1} u_k\, \omega_N^{-\ell k} = \sum_{k=0}^{N'-1} u_{2k}\, \omega_N^{-2k\ell} + \sum_{k=0}^{N'-1} u_{2k+1}\, \omega_N^{-(2k+1)\ell} \\
&= \sum_{k=0}^{N'-1} u_{2k} (\omega_N^2)^{-k\ell} + \sum_{k=0}^{N'-1} u_{2k+1}\, \omega_N^{-\ell} (\omega_N^2)^{-k\ell} \\
&= \hat{v}_\ell + \omega_N^{-\ell} \hat{w}_\ell \qquad \text{puisque } \omega_N^2 = \omega_{N'}
\end{aligned}$$

Transformée de Fourier rapide : Cooley-Tukey

Complexité

- Relation de récurrence $C(N) = 2C(N/2) + N$ produits complexes
- Si $N = 2^n$: on trouve $C(2^n) = n2^n$, d'où $C(N) = N\log_2(N)$ produits

Transformée de Fourier rapide : Cooley-Tukey

Complexité

- Relation de récurrence $C(N) = 2C(N/2) + N$ produits complexes
- Si $N = 2^n$: on trouve $C(2^n) = n2^n$, d'où $C(N) = N\log_2(N)$ produits

Remarques :

- même technique (et même complexité) pour la transformée inverse
- il faut aussi calculer les racines de l'unité ; ça reste en $O(N\log_2(N))$
- bonne stabilité numérique (transformation unitaire)
- application à $\mathbb{C}_D[X]$: **multiplication en** $O(D\log_2(D))$

Transformée de Fourier rapide : Cooley-Tukey

Complexité

- Relation de récurrence $C(N) = 2C(N/2) + N$ produits complexes
- Si $N = 2^n$: on trouve $C(2^n) = n2^n$, d'où $C(N) = N \log_2(N)$ produits

Remarques :

- même technique (et même complexité) pour la transformée inverse
- il faut aussi calculer les racines de l'unité ; ça reste en $O(N \log_2(N))$
- bonne stabilité numérique (transformation unitaire)
- application à $\mathbb{C}_D[X]$: **multiplication en** $O(D \log_2(D))$

⚠ Confusion fréquente entre *transformée de Fourier discrète* (application $u \mapsto \hat{u} \in \mathcal{L}(\mathbb{C}^N)$) et *transformée de Fourier rapide* (algorithme de calcul)

Division euclidienne rapide dans $K[X]$

Soient $A, B \in K[X]$, on note $a = \deg(A)$ et $b = \deg(B)$; o.p.s. $a \geqslant b$.

Division euclidienne : $A = BQ + R$ avec $\deg(R) < b$.
On sait $\deg(Q) = a - b$

Division euclidienne rapide dans $K[X]$

Soient $A, B \in K[X]$, on note $a = \deg(A)$ et $b = \deg(B)$; o.p.s. $a \geqslant b$.

Division euclidienne : $A = BQ + R$ avec $\deg(R) < b$.
On sait $\deg(Q) = a - b$

Idée 1 : $A = BQ$ "modulo" termes de petits degrés
$\rightsquigarrow$ les *polynômes réciproques* ramènent à congruence modulo X^{a-b+1}

$$\underbrace{X^a\, A(\frac{1}{X})}_{\tilde{A} \in K[X]} = \underbrace{X^b\, B(\frac{1}{X})}_{\tilde{B} \in K[X]} \times \underbrace{X^{a-b}\, Q(\frac{1}{X})}_{\tilde{Q} \in K[X]} + X^{a-b+1} \times \underbrace{X^{b-1}\, R(\frac{1}{X})}_{\in K[X]}$$

Division euclidienne rapide dans $K[X]$

Soient $A, B \in K[X]$, on note $a = \deg(A)$ et $b = \deg(B)$; o.p.s. $a \geqslant b$.

Division euclidienne : $A = BQ + R$ avec $\deg(R) < b$.
On sait $\deg(Q) = a - b$

Idée 1 : $A = BQ$ "modulo" termes de petits degrés
$\rightsquigarrow$ les *polynômes réciproques* ramènent à congruence modulo X^{a-b+1}

$$\underbrace{X^a A(\frac{1}{X})}_{\tilde{A} \in K[X]} = \underbrace{X^b B(\frac{1}{X})}_{\tilde{B} \in K[X]} \times \underbrace{X^{a-b} Q(\frac{1}{X})}_{\tilde{Q} \in K[X]} + X^{a-b+1} \times \underbrace{X^{b-1} R(\frac{1}{X})}_{\in K[X]}$$

- résoudre équation $\tilde{A} = \tilde{B}\tilde{Q} \mod X^{a-b+1}$
- récupérer Q
- calculer $R = A - BQ$

Division euclidienne rapide dans $K[X]$

Soient $A, B \in K[X]$, on note $a = \deg(A)$ et $b = \deg(B)$; o.p.s. $a \geqslant b$.

Division euclidienne : $A = BQ + R$ avec $\deg(R) < b$.
On sait $\deg(Q) = a - b$

Idée 1 : $A = BQ$ "modulo" termes de petits degrés
$\rightsquigarrow$ les *polynômes réciproques* ramènent à congruence modulo X^{a-b+1}

$$\underbrace{X^a A(\frac{1}{X})}_{\tilde{A} \in K[X]} = \underbrace{X^b B(\frac{1}{X})}_{\tilde{B} \in K[X]} \times \underbrace{X^{a-b} Q(\frac{1}{X})}_{\tilde{Q} \in K[X]} + X^{a-b+1} \times \underbrace{X^{b-1} R(\frac{1}{X})}_{\in K[X]}$$

- résoudre équation $\tilde{A} = \tilde{B}\tilde{Q} \mod X^{a-b+1}$ $\rightsquigarrow$ inversion rapide
- récupérer Q
- calculer $R = A - BQ$

Inversion modulo X^n : itération de Newton

Soit $P \in K[X]$ tel que $P(0) \neq 0$.

Idée 2 : calculer inverse de P modulo X^n par *approximations successives*

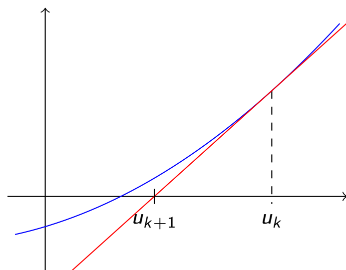
Inversion modulo X^n : itération de Newton

Soit $P \in K[X]$ tel que $P(0) \neq 0$.

Idée 2 : calculer inverse de P modulo X^n par *approximations successives*

Rappel : méthode de Newton pour résoudre $f(x) = 0$

- partir de u_0 solution approchée
- récurrence $u_{k+1} = u_k - \frac{f(u_k)}{f'(u_k)}$
- convergence quadratique sous de bonnes hypothèses



Inversion modulo X^n : itération de Newton (suite)

Adaptation aux polynômes

- résoudre $f(U) = 0 \ [X^n]$ avec $f : U \mapsto UP - 1$
- U_k solution approchée si $f(U_k) = 0 \ [X^m]$ pour un $m \leqslant n$
- on part de $U_0 = P(0)^{-1}$, vérifie $f(U_0) = 0 \ [X^1]$

Inversion modulo X^n : itération de Newton (suite)

Adaptation aux polynômes

- résoudre $f(U) = 0\ [X^n]$ avec $f : U \mapsto UP - 1$
- U_k solution approchée si $f(U_k) = 0\ [X^m]$ pour un $m \leqslant n$
- on part de $U_0 = P(0)^{-1}$, vérifie $f(U_0) = 0\ [X^1]$

Newton "classique" : $U_{k+1} = U_k - \frac{f(U_k)}{f'(U_k)} = U_k - \frac{U_k P - 1}{P}$

Mais : P^{-1} pas connu $\rightsquigarrow$ on remplace par approximation U_k.

Récurrence modifiée : $U_{k+1} = U_k - U_k(U_k P - 1)$

Inversion modulo X^n : itération de Newton (suite)

Adaptation aux polynômes

- résoudre $f(U) = 0\ [X^n]$ avec $f : U \mapsto UP - 1$
- U_k solution approchée si $f(U_k) = 0\ [X^m]$ pour un $m \leqslant n$
- on part de $U_0 = P(0)^{-1}$, vérifie $f(U_0) = 0\ [X^1]$

Newton "classique" : $U_{k+1} = U_k - \frac{f(U_k)}{f'(U_k)} = U_k - \frac{U_k P - 1}{P}$

Mais : P^{-1} pas connu $\rightsquigarrow$ on remplace par approximation U_k.

Récurrence modifiée : $U_{k+1} = U_k - U_k(U_k P - 1)$

Propriété

Si $U_k = P^{-1}\ [X^m]$, alors $U_{k+1} = P^{-1}\ [X^{2m}]$

Récurrence immédiate : $U_k = P^{-1}\ [X^{2^k}]$
("précision doublée" à chaque itération)

Inversion modulo X^n : complexité

Itération de Newton

- $U_0 = P(0)^{-1}$
- $U_k = U_{k-1} - U_{k-1}(U_{k-1}P - 1) = U_{k-1}(2 - U_{k-1}P)\ [X^{2^k}]$

Fin après $\lceil \log_2 n \rceil$ étapes

Réduction mod X^{2^k} essentiellement **gratuite**

À chaque étape : additions et deux produits de polynômes de degré $< 2^k$
$\rightsquigarrow O(k2^k)$ avec produit rapide (Fourier)

Inversion modulo X^n : complexité

Itération de Newton

- $U_0 = P(0)^{-1}$
- $U_k = U_{k-1} - U_{k-1}(U_{k-1}P - 1) = U_{k-1}(2 - U_{k-1}P)\ [X^{2^k}]$

Fin après $\lceil \log_2 n \rceil$ étapes

Réduction mod X^{2^k} essentiellement **gratuite**

À chaque étape : additions et deux produits de polynômes de degré $< 2^k$
$\rightsquigarrow O(k2^k)$ avec produit rapide (Fourier)

Coût total en $O\left(\sum_{k \leqslant \log_2 n} k2^k\right) = O(n \log n)$ opérations dans K

Remarque : intéressant même avec produit classique

Division euclidienne rapide : bilan

Pour $A = BQ + R$:

1. Prendre polynômes réciproques $\tilde{A}$ et $\tilde{B}$: zéro opération (retourner les coefficients)
2. Calculer $\tilde{B}^{-1} \bmod X^{a-b+1}$ par itération de Newton : en $O(a\log(a))$
3. Calculer $\tilde{Q} = \tilde{A}(\tilde{B})^{-1} \bmod X^{a-b+1}$: en $O(a\log(a))$
4. Prendre polynôme réciproque Q (gratuit) et calculer $R = A - BQ$: en $O(a\log a)$

Coût total en $O(a\log a)$.

Références

- Alin Bostan et al. : Algorithmes efficaces en calcul formel (Hal.inria.fr)
- Alexandre Casamayou et al. : Calcul mathématique avec Sage
- Victor Shoup : Computational introduction to number theory and algebra (Cambride University Press)
- **Cormen et al. : Introduction à l'algorithmique, cours et exercices (Dunod)**