TP2 : Complexité

Bibliographie :
— *Algorithmique*, Cormen, Leiserson, Rivest, Stein.
— *Cours de calcul formel*, Saux-Picard.

1 Avant de commencer

La complexité est un outil qui permet de mesurer l'efficacité d'un algorithme, indépendamment de la machine qui l'exécute. Elle permet de déterminer si un problème donné peut être résolu informatiquement (sous-entendu, en temps raisonnable !) et, dans le cas où plusieurs solutions sont proposées, laquelle est la plus rapide.

Bien sûr, la notion de "raisonnable" dépend du problème étudié, mais vous pouvez considérer, en première approximation, que les algorithmes pires que polynomiaux ne sont pas intéressants en pratique.

Temps	Type de complexité	Temps pour n = 5	Temps pour n = 50	Temps pour n = 250	Temps pour n = 1 000	Temps pour n = 10 000	Temps pour n = 1 000 000	Problème exemple
$O(1)$	complexité constante	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	Accès tableaux
$O(\log(n))$	complexité logarithmique	10 ns	20 ns	30 ns	30 ns	40 ns	60 ns	Recherche dichotomique
$O(n)$	complexité linéaire	50 ns	500 ns	2.5 μs	10 μs	100 μs	10 ms	Parcours de liste
$O(n\log(n))$	complexité linéarithmique	40 ns	850 ns	6 μs	30 μs	400 μs	60 ms	Tris dont le Tri fusion ou le Tri par tas
$O(n^2)$	complexité quadratique (polynomiale)	250 ns	25 μs	625 μs	10 ms	1 s	2.8 heures	Parcours de tableaux 2D
$O(n^3)$	complexité cubique (polynomiale)	1.25 μs	1.25 ms	156 ms	10 s	2.7 heures	316 ans	Multiplication matricielle non-optimisée.
$2^{\mathrm{poly}(n)}$	complexité exponentielle	320 ns	130 jours	10^{59} ans	...	...	...	Problème du sac à dos par force brute.
$O(n!)$	complexité factorielle	1.2 μs	10^{48} ans	...	...	...	...	Problème du voyageur de commerce (avec une approche naïve).

PIÈGE : La complexité s'exprime en fonction de la *taille de l'entrée*. Si votre problème consiste à trier une liste de n élements, il est naturel de l'exprimer en fonction de n. S'agissant d'un problème d'arithmétique/cryptographie, la taille de l'entrée pertinente est la longueur du mot binaire permettant de stocker les entiers de départ (et non les entiers eux-mêmes) !

♠ **Pour aller plus loin :** réduction de la complexité via programmation dynamique, problèmes NP, co-NP et NP-complets.

2 Mise en pratique : l'exponentiation

Premières expériences

1. Essayer de calculer brutalement `(3^400000000).mod(123)`.
2. Essayer avec `power_mod`.

Implémentation de l'algorithme d'exponentiation naïf

```
def expo_naive(x,n):
    result = 1
    for k in range(n) :
```

```
        result = result*x
    return result
```

3. Que fait cet algorithme ? Quelle est sa complexité (en fonction de n) ?

Implémentation de l'algorithme d'exponentiation rapide

```
def expo_rapide(x,n) :
    if n==0 :
        return 1
    else :
        if n%2==0 :
            return expo_rapide(x,n/2)^2
        else :
            return x*expo_rapide(x,(n-1)/2))^2
```

4. Prouver que cet algorithme calcule bien x^n.
5. Quelle est sa complexité ? Justifier.

Banc d'essai

Grâce à la commande `%time` placée en début de cellule, vous pouvez connaître le temps nécessaire au processeur pour effectuer votre calcul.

6. (Sur les entiers.) Donner un ordre de grandeur de la puissance à laquelle vous pouvez élever 3, sans que le calcul par `expo_naive` ne dure plus de dix secondes. Pour ces valeurs, combien de temps prend `expo_rapide` ? Jusqu'où peut monter `expo_rapide` en dix secondes ?
7. (Sur $\mathbb{Z}/n\mathbb{Z}$.) Mêmes questions pour l'élément 7 de $\mathbb{Z}/29\mathbb{Z}$.

```
♣ G=IntegerModRing(29), G(7)
```

8. (Sur $\mathbb{Q}[X]$.) Mêmes questions pour le polynôme $X + 1$.
9. (Sur $M_n(\mathbb{Q})$.) Mêmes questions pour la matrice $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$.

3 Calcul du n-ième terme de la suite de Fibonacci

On considère la suite de Fibonacci $F_{n+2} = F_{n+1} + F_n$ avec pour conditions initiales $F_0 = 0$ et $F_1 = 1$. Le but de cette partie est de comparer les performances de cinq algorithmes pour calculer le n-ième terme de la suite.

Récursif naïf

10. Implémenter l'algorithme `Fibo_recur` qui calcule récursivement F_n [1]. Le tester. Jusqu'à quel terme (environ) pouvez-vous monter en moins de deux secondes de calcul ?
11. Déterminer et justifier la complexité de l'algorithme. Comment l'améliorer ?

Itératif

12. Implémenter à présent `Fibo_iter`, une version itérative de l'algorithme (c'est-à-dire avec une boucle `for`). Seules trois variables de mémoire sont nécessaires. Tester. Jusqu'à quel terme (environ) pouvez-vous monter en moins de deux secondes de calcul ?
13. Déterminer et justifier la complexité de l'algorithme.

1. Un algorithme récursif est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème.

Récursif amélioré

14. Montrer que $\forall n, k \in \mathbb{N}, F(n+k+1) = F(k)F(n) + F(k+1)F(n+1)$.
15. Vous servir de cette observation pour écrire un algorithme récursif `Fibo_recur_mieux` plus performant que le premier. L'implémenter et tester ses temps de calcul.
16. Déterminer et justifier sa complexité.

Via exponentiation matricielle

17. Montrer que la suite des vecteurs $\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$ satisfait une équation récurrente d'ordre 1. En déduire l'expression du terme général de la suite.
18. En utilisant votre algorithme d'exponentiation naïve, écrire une fonction `Fibo_expo_matrice_naive` qui calcule F_n. La tester. Quelle est sa complexité ?
19. Même question en utilisant à présent votre algorithme d'exponentiation rapide.

Calcul "direct"

20. Calculer le terme général de la suite (F_n).
21. Etudier les performances de l'algorithme associé. Quelle peut être sa complexité ?