Complexité algorithmique

1) Complexité

La complexité en temps correspond au nombre d'opérations élémentaires éxécutées, la complexité en espace est le nombre de cases mémoires nécessaires au cours de l'algorithme. De façon générale, on ne distingue pas les coûts des opérations élémentaires (tests, opérations arithmétiques, affectation). C'est pourquoi on donne la complexité sous la forme $O(g(n))$.

Remarques : $O(n+m) = O(\max(n,m))$

Remarque : La notion d'opération élémentaire n'est très bien définie (à moins de revenir aux opérations sur les bits binaires). Par exemple, si on considère uniquement les opérations sur les bits 0-1, la somme de deux grands entiers est de complexité *linéaire* en la représentation binaire de ces entiers (c'est pourquoi l'algorithme d'exponentiation rapide n'est pas plus efficace que l'algorithme naïf quand on l'applique aux entiers naturels ...). En revanche l'algorithme d'exponentiation rapide est très efficace si on l'applique aux produits de matrices flottantes (ou booléennes), car le produit de deux matrices flottantes ne dépend pas de la taille de ses coefficients..

- Pour évaluer la complexité, considérer le nombre d'appels, d'empilements, etc ...

- Complexité dans le pire des cas, en moyenne, dans le meilleur cas.

Exemple : La complexité du tri par pivot est en $O(n \ln n)$ en moyenne et en $O(n^2)$ dans le pire des cas.

Remarque : Notation Θ : On a $f(n) = \Theta(g(n))$ ssi $f(n) = O(g(n))$ et $g(n) = O(f(n))$.
Ainsi, $f(n) = \Theta(n^2)$ ssi il existe a et $b > 0$ tels que pour n assez grand, $an^2 \leq f(n) \leq bn^2$.

Classes P et NP (*complément culturel*)

- *Problèmes de classe P* : on connaît un algorithme polynomial en la longueur des données.

Exemples de problèmes de classe P : recherche du maximum, algorithmes de tri , test de connexité dans un graphe et existence d'un cycle eulérien (circuit passant exactement une fois par chaque arête d'un graphe).

- *Problèmes de classe NP* : ("polynomiaux non-déterministes") : les meilleurs algorithmes ne sont pas nécessairement polynomiaux, mais toute solution proposée peut être vérifiée en temps polynomial : existence d'un cycle hamiltonien (cycle passant exactement une fois par chaque sommet d'un graphe), problème du voyageur de commerce, problème du sac-à-dos.

Remarque : Le terme *NP* ne signifie pas "non polynomial", même si jusqu'à présent, aucun algorithme polynomial ne permet de résoudre tous les problèmes *NP*.

2) Mémoïsation

La complexité en temps peut souvent être diminuée en augmentant la complexité en espace : le principe général, appelé "mémoïsation", **consiste à stocker des informations pour éviter des calculs redondants.**

Il est souvent judicieux d'effectuer les différentes opérations à faire dans un ordre judicieux de sorte à exploiter les résultats précédents dans le calcul en cours.

Dans les algorithmes utilisant le principe de la programmation dynamique (procédant par évaluations de sous-structures, disjointes ou non), on utilise généralement la **mémoïsation**.

Exemple : Calcul du n-ième de la suite de Fibonacci : $u_{n+2} = u_{n+1} + u_n$.

On mémorise (u_n, u_{n+1}) qu'on met à jour à chaque étape en $O(1)$ opérations. Donc complexité en $O(n)$.

3) Calculs de complexité dans les algorithmes utilisant des boucles for

a) *Exemple* :

```
for  i in range(n) :  instruction(i)
for j in range(m) :  instruction(j)
```

Complexité en $O(n+m)) = O(\max(n,m))$

b) *Exemple* :

```
for  i in range(n) :
    instruction(i)
     for j in range(m) :  instruction(i,j)
```

Complexité en $O(nm)$.

Remarque : Si chaque indice dans les différentes `for` décrit un intervalle en $O(n)$, on obtient toujours une complexité polynomiale en n (et jamais exponentielle).

c) *Exemple* : Crible d'Eratosthène

```
premiers = [1]*n ; premiers[0] = 0 ; premiers[1] = 0
for p in range(2,n) :
    if premiers[p] = 1 :
          k = 2
         while k*p < n :  premiers[k*p] = 0 :  k = k+1
```

La complexité est en $O(m)$, où $m = \sum_{p \text{ premier et } p<n} \frac{n}{p} \leq n \sum_{k=2}^{n} \frac{1}{k} \sim n \ln n$.

4) Calculs de complexité dans les algorithmes utilisant des boucles while

Méthodes généralement utilisées :

- Utiliser une suite décroissante (principe de Fermat) pour évaluer le nombre d'itérations.

- Compter combien fois un objet (judicieusement choisi) intervient dans la boucle.

a) *Exemple* : Intersection de listes triées par ordre croissant

```
01.    def inter(A,B) :
02.         n = len(A) ; m = len(B) ; i = 0 , j = 0 ; L = []
03.         while i<n and j<m :
04.              if A[i] == B[j] :  L.append(A[i]) ; i = i+1 ; j = j+1
05.              elif A[i] < B[j] :  i = i+1
06.              else :  j = j+1
```

On peut noter qu'un même élément `A[i]` peut être utilisé un grand nombre de fois.

En revanche, $i+j$ augmente d'au moins 1 à chaque itération. D'où la complexité $O(n+m)$.

b) *Exemple* : On considère L une liste de n listes codant une partition de $\{0,1,...,N-1\}$.

Autrement, les listes $L[i]$ sont disjointes et la somme de leurs longueurs vaut N.

```
01.     M = [0]*n
02.     for i in range(n) :
03.          for k in L[i]
04.               if test(k) :  M[i] = 1
```

Ainsi, à la sortie, $M[i]$ vaut 1 ssi la i-ième liste $L[i]$ contient au moins un élément k vérifiant le test.

Notons p_i la longueur de la liste $L[i]$. La complexité est en $O(m)$, où $m = n + \sum_{i=0}^{n-1} p_i = n + N$.

c) (★) Calcul du pgcd par divisions euclidiennes successives

Soient des entiers $a \geq b$. On considère (x_n) la suite ainsi calculée : $x_0 = a$, $x_1 = b$ et tant que x_n n'est pas nul, on définit $x_{n+1} = x_{n-1} \bmod x_n$. L'algorithme termine car (x_n) décroît strictement.

On a $\text{pgcd}(a,b) = \text{pgcd}(x_0,x_1) = \text{pgcd}(x_1,x_2) = ... = \text{pgcd}(x_m,x_{m+1}) = x_m$ où m est tel que $x_{m+1} = 0$.

```
def pgcd(a,b) :
     x = a ; y = b
     while y > 0 :
         r = x % y ; x = y ; y = r    # variante :  x,y = y,x%y
     return x
```

On a $x_{n+1} \leq x_{n-1} - x_n$ (les quotients sont ≥ 1), ce qui s'écrit aussi $x_{n-1} \geq x_n + x_{n+1}$.

Par comparaison avec une suite de Fibonacci, on obtient $x_n \geq \alpha\varphi^{m-n} + \beta$, où $\varphi = \frac{1}{2}(1+\sqrt{5})$ et $\alpha > 0$.

Donc le nombre m d'étapes est en $O(\log a)$, car $x_0 = a$.

5) Calculs de complexité par une formule de récurrence

a) **Principe général :** On trouve une relation de récurrence (souvent forte) vérifiée par $c(n)$, où n est la taille (ou un paramètre) de la structure entrée comme argument de la fonction récursive.

Par exemple, si $c(n) \leq c(n-1) + O(n^\alpha)$, alors il existe K tel que $\forall n \geq 1$, $c(n) \leq c(n-1) + Kn^\alpha$.

Donc $c(n) \leq c(0) + K\sum_{j=1}^{n} j^\alpha = O(n^{\alpha+1})$.

b) **Cas classique de type dichotomie** ("diviser pour régner") : $c(n) \leq ac(\lceil \frac{n}{2} \rceil) + bc(\lfloor \frac{n}{2} \rfloor) + f(n)$.

Exemple : Recherche par dichotomie dans une liste triée.

On obtient $c(n) \leq c(\lceil \frac{n}{2} \rceil) + O(1)$.

Supposons $n = 2^p$. Alors $c(n) \leq c(\frac{n}{2}) + K \leq c(\frac{n}{4}) + 2K \leq ... \leq c(1) + pK = O(\log n)$.

Exemple : Recherche du k-ième élément (sélection) dans le cas où le pivot est en position idéale.

On obtient $c(n) \leq c\left(\lfloor \frac{n}{2} \rfloor\right) + Kn$. Supposons $n = 2^p$.

Alors $c(n) = c(\frac{n}{4}) + K\frac{n}{2} + Kn = ... = c(1) + Kn(1 + \frac{1}{2} + \frac{1}{4} + ...) \leq c(1) + 2Kn = O(n)$.

Exemple important : Tri par fusions successives de listes triées.

On obtient $c(n) = c(\lfloor \frac{n}{2} \rfloor) + c(\lceil \frac{n}{2} \rceil) + Kn$.

Supposons $n = 2^p$. Alors $c(n) \leq 2c(\frac{n}{2}) + Kn \leq 4c(\frac{n}{4}) + Kn + 2K(\frac{n}{2}) = 4c(\frac{n}{4}) + 3Kn$.

En itérant p fois, on obtient $c(n) \leq 2^p c(\frac{n}{2^p}) + pKn = 2^p c(1) + pKn = O(n \log n)$.

Remarque : Dans les exemples précédents, on a calculé la complexité uniquement dans les cas où $n = 2^p$.
Pour le cas où n est arbitraire, on utilise souvent l'argument suivant : en général, $n \longmapsto c(n)$ est croissante. On considère donc p tel que $2^{p-1} < n \leq 2^p$, donc $c(n) \leq c(2^p)$.
On obtient une majoration du coût en fonction de 2^p, et on conclut en notant que $2^p \leq (2n)$.

c) (★) **Exemples de calculs de complexité moyenne** (*complément*)

- Dans le paragraphe b), on a évalué $c(n)$ pour $n = 2^p$ en explicitant un majorant par applications successives de la relation de récurrence. Ici, on va utiliser une autre méthode : pour prouver que $c(n) = O(n)$, on va chercher une condition sur a de sorte que la relation $c(n) \leq an$ soit compatibale avec la relation de récurrence : si $c(k) \leq ak$ pour tout entier $k < n$, alors on a $c(n) \leq an$. Dans ce cas, il suffit de vérifier que la relation est vraie pour les premiers termes (en prenant a assez grand).

- *Exemple* : Recherche du k-ième élément (sélection) avec un pivot choisi aléatoirement.

On a $c(0) = 0$ et $\forall n \geq 1$, $c(n) = Kn + \dfrac{1}{n}\sum_{k=0}^{n-1} c(k)$.

Montrons qu'il existe a tel que $\forall n \in \mathbb{N}$, $c(n) \leq an$.

Supposons $n \geq 1$ et la propriété vraie $c(k) \leq ak$ jusqu'au rang $n-1$.

Donc $\sum_{k=0}^{n-1} c(k) \leq a\sum_{k=0}^{n-1} k \leq \frac{1}{2}an^2$, donc $c(n) \leq (\frac{1}{2}a + K)n$.

On choisit a de sorte que $\frac{1}{2}a + K \leq a$, par exemple $a = 2K$. Alors $\forall n \in \mathbb{N}$, $c(n) \leq 2Kn$.

- *Exemple* : Complexité moyenne de Quicksort (avec un pivot choisi aléatoirement).

On a $c(0) = 0$ et $\forall n \geq 1$, $c(n) = Kn + \dfrac{1}{n}\sum_{k=0}^{n-1}(c(k) + c(n-1-k)) = Kn + \dfrac{2}{n}\sum_{k=0}^{n-1} c(k)$.

On va montrer qu'il existe a tels que $\forall n \in \mathbb{N}$, $c(n) \leq an \ln n$.

Supposons $n \geq 1$ et la propriété vraie jusqu'au rang $n-1$.

Donc $\sum_{k=0}^{n-1} c(k) \leq a\sum_{k=0}^{n-1} k \ln k \leq a\int_1^n t \ln t \; dt = \frac{1}{2}an^2 \ln n - \frac{1}{4}a(n^2-1)$.

Donc $c(n) \leq an \ln n + Kn - \frac{1}{4}a(n^2-1)$.

Le discriminant de la fonction polynôme $n \longmapsto Kn - \frac{1}{4}a(n^2-1)$ vaut $K^2 - a$.

On choisit a tel que $\Delta > 0$, par exemple $a = \sqrt{K}$. Alors $\forall n \in \mathbb{N}$, $Kn - \frac{1}{4}a(n^2-1) < 0$.

Ainsi par récurrence forte, on a $\forall n \in \mathbb{N}$, $c(n) \leq an \ln n$.