

Introduction à la Complexité

TD complexité

Alexandre Saidi

ECL 2A - LIRIS

Spetembre 2022

Rappels : Complexité d'un algorithme ?

- Estimation des ressources nécessaires à l'exécution
 - Complexité en temps = estimation du nombre d'instructions
 - Complexité en espace = estimation de l'espace mémoire
 - ➔ comparer 2 algorithmes étant donné la taille n des données en entrées
- Ordre de grandeur O
$$O(f(n)) \rightarrow \exists c, n_0 | \forall n > n_0, \text{nbr. instructions carac.} < c.f(n)$$
 - $O(\log(n))$: logarithmique
 - $O(n)$: linéaire
 - $O(n^k)$: polynomial
 - $O(kn)$: exponentiel

Exemples de complexité d'algorithmes

- Rechercher un élément dans un tableau de n éléments
 - Algorithme de recherche séquentielle : $O(n)$
 - Algorithme de recherche dichotomique / tableau trié : $O(\log(n))$
- Maximiser une fonction linéaire sous des contraintes linéaires
 - Algorithme du simplexe : $O(2^n)$ dans le pire des cas...
 - faiblement polynomial en pratique !
 - Algorithme du point intérieur : polynomial
- Chercher un circuit hamiltonien dans un graphe
 - Algorithme énumérant toutes les permutations des sommets : $O(n!)$

Exemples de complexité d'algorithmes (suite)

- Trier un tableau de n éléments : quelques méthodes
 - Algorithme de tri par **sélection** : $O(n^2)$
 - Algorithme de tri par **insertion** :
 - dans le meilleur des cas (tableau déjà trié) : $O(n)$
 - dans le pire des cas (tableau trié à l'envers) : $O(n^2)$
 - Algorithme de tri **rapide** (quicksort) :
 - dans le meilleur des cas (pivot = médiane) : $O(n \log(n))$
 - dans le pire des cas (pivot = élt. max ou min) : $O(n^2)$
 - Algorithme de tri par **tas** : $O(n \log(n))$
 - Algorithme de tri **fusion** : $O(n \log(n))$

Calculs : quelques rappels

- Appels séquentiels :

```
f(...) :  
  g1(...)  
  g2(...)  
  ...  
  gk(...)
```

$$\rightarrow \text{cout}(f(\dots)) = \sum \text{cout}(g_i)$$

- Appels conditionnels :

```
f(...) :  
  if cond1 :  
    g1(...)  
  elif cond2 :  
    g2(...)  
  ...  
  gk(...)
```

$$\rightarrow \text{cout}(f(\dots)) = \max(\text{cout}(g_i)) \text{ (on néglige l'évaluation de } \text{cond}_j)$$

Calculs : quelques rappels (suite)

- Itérations :

```
f(..., n) :  
  for i in range(n) :  
    g(...)
```

$$\rightarrow \text{cout}(f(\dots)) = n.\text{cout}(g)$$

- ☞ Si l'appel de g dépend de i de la forme $g(\dots, i)$:
alors on préfère sommer les complexités :

$$\rightarrow \text{cout}(f(\dots, n)) = \sum_i^n \text{cout}(g(\dots, i))$$

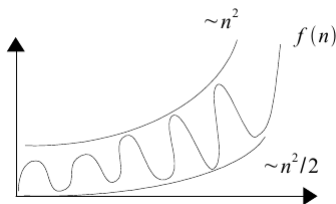
Calculs : quelques rappels (suite)

- Rappel de la complexité en O en lien avec Ω :

$T(n)$ est $O(f(n))$ noté $T(n) = O(f(n))$ et $T(n) = \Omega(g(n))$ si :

$$\exists N, \alpha, \beta | \forall n > N, \alpha.f(n) \leq T(n) \leq \beta.g(n)$$

- On évitera les écriture inutiles telles que $x^2 = O(x^3)$.
→ Il n'est pas toujours utile d'estimer le coût tel que $T(n) = n^2$ en $T(n) = O(n^3)$.
- La définition donnée ci-dessus permet d'encadrer la complexité comme dans :



Complexité : quelques équivalences

Règles de ré-écriture :

- $f(n) = O(f(n))$
- $g = O(f(n)), f = O(h(n)) \implies g = O(h(n))$
- $c.O(g(n)) = O(c.g(n)) = O(g(n))$
- $f(n).O(g(n)) = O(f(n).g(n))$
- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
pour la commodité des écritures !

Complexité : quelques équivalences (suite)

- **Un exemple d'utilisation :**

sachant $f(n) = 5.n^2 + 6.\log(n) + 7$

et $g(n) = \sqrt{n} + \log(n)$,

calculer $O(f(n).g(n))$.

◦ Par $f(n) = 5.n^2 + 6.\log(n) + 7$, on peut écrire $f(n) = O(n^2)$

et de $g(n) = \sqrt{n} + \log(n)$, on peut écrire $g(n) = O(n^{0.5})$

Et donc :

$$O(f(n).g(n)) = O(n^2.n^{0.5})$$

☞ Pas d'exercice pour cette partie !

Comparaison symbolique de complexité

Pour la comparaison (symbolique) de complexités, on dispose de quelques autres outils.

- Outils "limites" :

$$\lim_{x \rightarrow +\infty} \frac{x^{\beta > 0}}{e^x} \rightarrow 0 \quad \lim_{x \rightarrow +\infty} \frac{\log x^{\alpha > 0}}{x^{\beta > 0}} \rightarrow 0 \quad x^y = e^{y \cdot \ln x} \quad \log x = \frac{\ln x}{\ln 10} \quad \ln_2 = \frac{\ln x}{\ln 2} = \frac{\log x}{\log 2}$$

- Par exemple, ces outils nous serviront à trier l'ensemble de classes de complexité de la forme $O(n^\alpha \cdot \log^\beta n)$.

Ici, on peut rapidement obtenir $O(n^{0.5}) < O(n) < O(n^2)$

Et en insérant les éléments logarithmiques, on aura :

$$O(n^{0.5}) < \underbrace{O(n^{0.5} \cdot \log n)} < O(n) < \underbrace{O(n \cdot \log n)} < O(n^{1.5}) < \underbrace{O(n^{1.5} \cdot \log n)} < O(n^2)$$

Comparaison symbolique de complexité (suite)

Rappel des propriétés des limites :

◦ $\lim_{\infty} \frac{g(n)}{f(n)} \rightarrow 0$: $g(n)$ est d'ordre inférieur à $f(n)$ et

$$O(f(n) + g(n)) = O(f(n)) \quad (\text{on note } O(g(n)) < O(f(n)))$$

◦ $\lim_{\infty} \frac{g(n)}{f(n)} \rightarrow \infty$: $g(n)$ est d'ordre supérieur à $f(n)$ et

$$O(f(n) + g(n)) = O(g(n))$$

◦ $\lim_{\infty} \frac{g(n)}{f(n)} \rightarrow c > 0$: $g(n)$ est du même ordre que $f(n)$ et

$$g(n) = \Theta(f(n)) \text{ (et inversement)}$$

Exercices de vérification de complexité

- Proposer (sans calcul) la complexité des algorithmes suivants :
 - La recherche dichotomique d'un élément dans un tableau ordonné de taille n est?
 - Dans les deux versions itérative et récursive
 - récursive .
 - La recherche d'un élément dans une liste de taille n est?
 - La recherche du minimum / maximum dans une liste de taille n est?
 - La fusion de deux listes triées (de taille n et m) est?

Exercices complexité

Estimer (sans calcul) l'expression des complexité $O(\cdot)$ des récurrences suivantes :

- $n^2 + 100n - 7$
- $5n^4 + 2n^2 + 100$
- $5n + 8n^2 + 100n^3$
- $n\log_3 n + n\log_2 n$
- $2n^3 + 0.01n^2 - 5$
- $n^2 + n - \log_2(2^{n^2})$
- $5 + 0.001n^3 + 0.025n$
- $500n + 100n^{3/2} + 50n\log_{10} n$
- $0.3n + 5n^{1.5} + 2.5n^{1.75}$
- $100n + 0.01n^2$
- $100n^2 + 0.01n$
- $\log n + \log(\log(n))$
- $n^3 - n^2$ (le rôle du terme négatif)

Rappels du cours

☞ Avant de proposer des exercices sur la démonstration d'une complexité, quelques exemples de calculs du cours sont rappelés (voir cours sec. XII-1 pour d'autres exemples):

(I) Montrer que $n^2 + 2n + 1$ est $O(n^2)$

→ Il faut montrer : $n^2 + 2n + 1 \leq c.n^2$ avec $n \geq n_0 \geq 1$ entier positif.

Posons $n_0 = 1$

$n^2 + 2n + 1 \leq cn^2$ avec $n \geq 1$. Divisons par n^2

$(n^2 + 2n + 1)/n^2 \leq c$ quand $n \geq 1$.

Sachant que $(n^2 + 2n + 1)/n^2 \leq (n^2 + 2n^2 + n^2)/n^2$ avec $n \geq 1$,

on cherche c tq $\boxed{(n^2 + 2n + 1)/n^2 \leq (n^2 + 2n^2 + n^2)/n^2 \leq c}$ avec $n \geq 1$.

→ Calcul de c : de la partie $(n^2 + 2n^2 + n^2)/n^2 \leq c$ lorsque $n \geq 1$.

on simplifie sur n^2 : $(1 + 2 + 1)/1 \leq c$ pour $n \geq 1$

→ $4 \leq c$ et $n \geq 1$

D'où : $n^2 + 2n + 1 \leq c.n^2$ pour (p.ex.) $c = 4, n \geq 1$.

Rappels du cours (suite)

(II) Montrer que $n^2/2 - 3n$ est $\Theta(n^2)$.

On doit montrer que : $\boxed{c_1 n^2 \leq n^2/2 - 3n \leq c_2 n^2}$ pour tout $n \geq n_0$.

→ On divise par n^2 :

$$c_1 \leq 1/2 - 3/n \leq c_2$$

→ Pour $c_2 \geq 1/2$, on a l'inégalité à droite.

→ Pour la partie gauche, $n \geq 7$ et $c_1 \leq 1/14$.

Donc, avec $n_0 = 7$, $c_1 = 1/14$ et $c_2 = 1/2$, nous aurons

$$T(n^2/2 - 3n) = \Theta(n^2).$$

☞ On peut montrer une complexité $\Theta(n)$ en démontrant une complexité $O(n)$ et $\Omega(n)$.

Exercices de preuve de complexité

- Prouver (par la définition de $O(\cdot)$ en trouvant les constantes) les complexités suivantes?
 - $2n^3 + 1 = O(n^3)$
 - $3n^3 - 4n^2 + n + 50 = O(n^3)$
- (Piège) : calculer la complexité de la fonction suivante (pour n très grand):

```
def f(n):  
    s=0  
    for i in range(0,n+1) :  
        for j in range(i,n+1)  
            s = s+1
```


Exercices combinaison complexités

(I) On a l'affirmation "**la complexité de tel algorithme est au moins $O(n^2)$** ".

→ Qu'est-ce que cette affirmation peut vouloir dire ?

(II) Soient $f(n)$ et $g(n)$ deux fonctions asymptotiques non négatives. En utilisant la définition de $\theta(\cdot)$ montrez que $\max(f(n), g(n)) = \theta(f(n) + g(n))$.

(III) Montrer que pour tous réels a, b avec $b > 0$, on a $(n + a)^b = \theta(n^b)$.

(IV) Prouver :

$$\circ 2^{n+1} = O(2^n)$$

Exercices sur les limites

- En utilisant les propriétés des limites et en particulier :

Si $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, alors $f = o(g)$ et f est d'ordre inférieur à g .

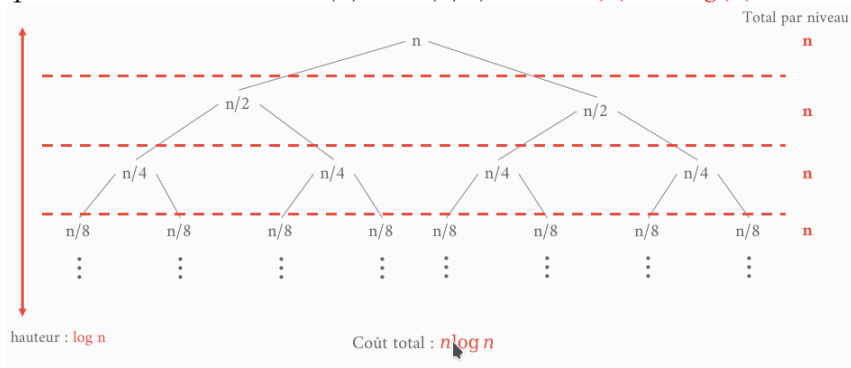
Montrez les complexités little-oh suivantes :

- $\sqrt{n} = o(n)$
- $n = o(n \cdot \log(\log(n)))$
- $n \cdot \log(\log(n)) = o(n \cdot \log(n))$
- $n \cdot \log(n) = o(n^2)$
- $n^2 = o(n^3)$

Ex. de Calcul par arbre de récursivité

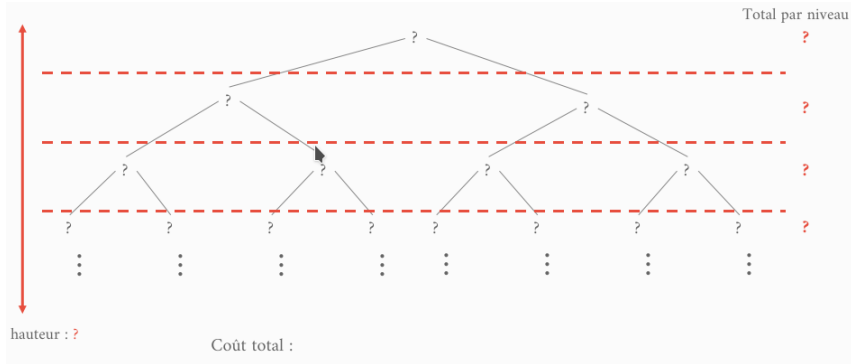
Une méthode d'estimation de la complexité : **arbre de récursion**

- Exemple : observation de l'arbre de récursivité du **Tri fusion** dont l'équation de récurrence est $T(n) = 2T(n/2) + n \rightarrow T(n) = n.\log(n)$



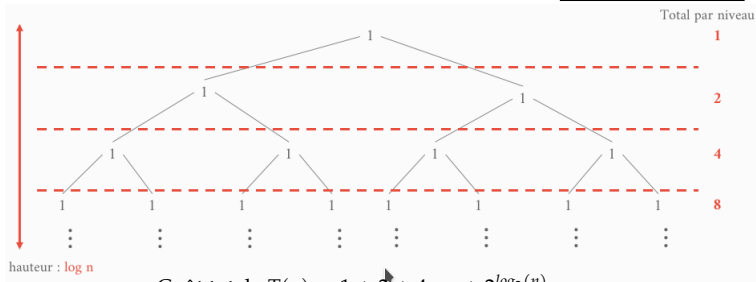
Ex. de Calcul par arbre de récursivité (suite)

- Exemple : recherche récursive du maximum dans le tableau non trié avec l'équation de récurrence $T(n) = 2T(n/2) + 1$



Ex. de Calcul par arbre de récursivité (suite)

- Solution : recherche récursive du maximum dans le tableau non trié :



→ Sachant que $\sum_{k=0}^n 2^k = 2^{n+1} - 1$, et que $2^{\log_2(n)} \simeq n$, on aura :

$$T(n) = \sum_{k=0}^{\log_2(n)} 2^k = 2^{\log_2(n)+1} - 1 = 2n - 1 \text{ et donc } T(n) = O(n)$$

Ex. de Calcul par substitution

- On prend l'exemple du Tri Fusion : $T(n) = 2T(n/2) + n$

- Par la technique de substitution, on a :

$$T(n) = 2T(n/2) + n$$

$$= 2(2T(n/4) + n/2) + n$$

$$= 4T(n/4) + n + n$$

$$= 8T(n/8) + n + n + n$$

$$= \dots$$

$$= n.T(n/n) + n + \dots + n$$

$$= n + n + \dots + n \quad \leftarrow \log(n) + 1 \text{ fois}$$

$$= n.(\log(n) + 1) = n.\log(n) + n \leq 2n\log(n) = O(n\log(n))$$

Exercices via l'arbre ou par substitution

Utiliser l'arbre de récursion ou la substitution pour définir la complexité de l'équation de récurrence :

- $T(n) = 3T(n/2) + n$
- $T(n) = T(n/2) + n^2$
- $T(n) = 4T(n/2) + cn$
- $T(n) = T(n-a) + T(a) + cn$ avec $a \leq 1, c > 0$ constantes.
- $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$ avec $0 < \alpha < 1$ et $c > 0$ sont constantes.

Vérifier votre solution par une méthode de substitution.

- A l'aide de l'arbre de récursion, vérifier que la solution à l'équation de récurrence $T(n) = T(n/3) + T(2n/3) + cn$, où c est une constante est $\Omega(n \cdot \log n)$.

Ex : Recherche des constantes de O, Ω, Θ

Un exemple : Tri Fusion

Par l'algorithme récursif de Tri-Fusion, nous avons l'équation de récurrence :

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Admettons que cette complexité est $T(n) = O(n \cdot \log(n))$

→ Peut-on prouver (par la déf. de $O(\cdot)$) : $T(n) \leq c \cdot n \cdot \log(n)$ pour une cst. $c > 0$?

Preuve :

◦ Hypothèse de récurrence : $T(m) = O(m \cdot \log(m)) \quad \forall m < n$
 en particulier, pour $m = \frac{n}{2}$, on a : $T\left(\frac{n}{2}\right) = O\left(\frac{n}{2} \log\left(\frac{n}{2}\right)\right)$

◦ On substitue dans l'expression de $T(n)$:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad \text{On remplace } T\left(\frac{n}{2}\right)$$

$$\leq 2c \cdot \frac{n}{2} \log\left(\frac{n}{2}\right) + n$$

$$= c \cdot n \cdot \log\frac{n}{2} + n$$

$$= c \cdot n \cdot \log(n) - c \cdot n \cdot \log(2) + n$$

$$= c \cdot n \cdot \log(n) - c \cdot n + n$$

$$\boxed{T(n) \leq c \cdot n \cdot \log(n) \quad \forall c \geq 1}$$

poser p.ex. $c=1$

CQFD.

Ex : Recherche des constantes de O , Ω , Θ

(suite)

Dans l'algorithme de Tri Fusion, on a :

cas de base : $T(1) = 0$

cas récurrent : $T(n) \leq c.n \log(n)$, $\forall c \geq 1$

A partir de ces deux cas, peut-on prouver :

$T(2n) \leq c.2n.\log(2n)$ pour $n \geq 1$?

Preuve (on suppose $n = 2^k$) :

$$T(2n) \leq c.2n.\log(2n) = 2c.n(1 + \log(n)) = 2c.n.\log(n) + 2c.n \text{ pour } c \geq 1 \quad (1)$$

Aussi, par l'équation de récurrence de la Tri Fusion, on a $T(n) = 2T(n/2) + n$

$$\rightarrow T(2n) = 2T\left(\frac{2n}{2}\right) + 2n$$

$$T(2n) = 2T(n) + 2n \quad \text{on remplace } T(n) \leq c.n \log(n)$$

$$T(2n) \leq 2.c.n \log(n) + 2n \quad \text{par le cas récurrent}$$

$$T(2n) \leq 2.c.n \log(n) + 2c.n \text{ pour } c \geq 1 \quad (2)$$

\rightarrow On a (1) et (2) qui coïncident.

Exercices de recherche des constantes

- Prouver la complexité de l'algorithme de recherche dichotomique.
 - Poser l'équation de récurrence
 - Démontrer que la complexité est $O(\log n)$ en recherchant les constantes des fonctions de croissance.
- De même pour les cas suivants : faire une hypothèse puis prouver en recherchant les constantes des fonctions O , Ω , Θ :
 - $T(n) = T(n-1) + n$ est $O(n^2)$.
 - $T(n) = T(n/2) + 1$ est $O(\log n)$
 - $T(n) = 2T(n/2) + 17 + n$ est $O(\log n)$
 - $T(n) = 2T(n/2) + n$ est $O(n \log n)$.

Montrer aussi que cette complexité est également $\Omega(n \log n)$. En conclure que la complexité est $\Theta(n \log n)$

Exercice : induction

- Prouver par induction la complexité de la recherche séquentielle est $O(n)$:
 - Cas de base : $T(1) = 1$
 - Cas de récurrence : $T(n) = T(n - 1) + 1$
- Prouver par induction la complexité de merge-sort décrite ci-dessous est $O(n.\log(n))$:
 - Cas de base : $T(2) = 2$
 - Cas de récurrence : $T(n) = 2T(n/2) + n$

Induction : l'importance de cas de base

Remarques :

Le "cas de base" (ou "base" tout court) est le cas non-récuratif, se dit également la "condition d'arrêt".

☞ Dans le raisonnement par récurrence, ne pas oublier le cas de base !

- Non seulement dans l'algorithme, mais aussi dans la complexité.
 - ➔ on doit vérifier que le/s cas de base satisfait/ont la relation trouvée.



Induction : l'importance de cas de base (suite)

Cas de base dan Tri Fusion :

Dans Tri-Fusion avec la récurrence $T(n) = 2T(n/2) + n$:

proposons $T(1) = 1$

→ Est-ce correct?

Non. On a un problème :

Car avec $T(n) \leq c.n.\log(n) \rightarrow T(1) \leq c.1.\log(1) = 0$

→ **Impossible** de trouver $c > 0$!

Induction : l'importance de cas de base (suite)

Rappel :

- On veut prouver $T(n) \leq c.n.\log(n) \forall n \geq n_0 \dots$ et on peut choisir le n_0 qu'on veut!

☞ Notre point est : que proposer pour n_0 tq. $\forall n > n_0 \dots$

→ $n_0 = 2$? puisque $n_0 = 1$ ne convient pas!

- Vérifions :

par l'équation de récurrence, on a $T(2) = 2.T(1) + 2$

→ $T(2) = 2.T(1) + 2 \rightarrow T(2) = 2 + 2 = 4$

→ Il reste donc à trouver une constante c t.q. $T(2) = 4 \leq 2c$

→ $c = 2$ convient :

$\forall n \geq 2, T(n) \leq 2.n.\log(n) = O(n.\log(n))$ (ça nous limite à $n_0 = 2$!)

Induction : l'importance de cas de base (suite)

- ☞ avec $n_0 = 2$, nous ne pourrions pas exécuter notre algorithme de tri avec un tableau de taille 1 ?
- ☞ **Mieux** et plus logique :
 - si on pose $T(1) = 0 \rightarrow T(2) = 2.T(1) + 2 = 2 \dots$
 - **$c = 2$ convient toujours**
- ☞ Le cas de base intervient grandement dans le calcul de $T(n)$,
Donc, c'est important.

Exercices éq de récurrence homogènes

- Montrez que la complexité de $\text{Fib}(n)$ est entre $\Omega((\frac{3}{2})^n)$ et $O((\frac{5}{3})^n)$.
- Calculer la complexité de $\text{Hanoi}(n)$ par son équation de récurrence homogène.
- Soit la récurrence :

$$t_n = 7t_{n/2} \quad \text{pour } n > 1, n \text{ est une puissance de } 2$$

$$t_1 = 1$$

Montrer que la complexité de ce problème est $t_n = O(7^{\log n})$

N.B. : on sait $7^{\log n} = n^{\log(7)} \approx n^{2.81}$

Exercices éq. de récurrence non homogènes

☞ Choisir 3 exercices parmi ci-dessous.

Trouver une solution générale pour les équation non homogènes suivantes :

- $a_n = 2a_{n-1} + 3(2^n)$, $a_0 = 1$
- $T(n) = 2T(n/4) + n$
- $T(n) = 4T(n/4) + n$
- $T(n) = 5T(n/4) + n$
- $T(n) = 16T(n/4) + n$
- $T(n) = 16T(n/4) + n^2$
- $T(n) = 3T(2n/3) + 1$ (algorithme Stooge-sort)
- $T(n) = T(n/2^d) + d^2n^{1/d}$ (récursion sur une mesh d-dimensionnelle)
- $T(n) = T(n/2) + \log(n)$ (Parallèle merge-sort)

Complexité et Master Theorm

Rappel du "Master Therorm" : on suppose donnée la relation de récurrence de la forme :

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \text{ avec } a > 1, b > 1.$$

où f est une fonction à valeurs entières positives.

Le "master theorem" s'énonce comme suit :

1- Si $f(n) = O(n^c)$ avec $c < \log_b a$

$$\text{alors } T(n) = \Theta(n^{\log_b a}).$$

2- Si $f(n) = \Theta(n^c \log^k n)$ avec $c = \log_b a$ et une constante $k \geq 0$

$$\text{alors } T(n) = \Theta(n^c \log^{k+1} n).$$

3- Si $f(n) = \Omega(n^c)$ avec $c > \log_b a$

et s'il existe une constante $k < 1$ telle que, pour n assez grand, on a : $af\left(\frac{n}{b}\right) \leq kf(n)$

(cette condition est appelée parfois la "condition de régularité"),

alors on a : $T(n) = \Theta(f(n))$.

Complexité et Master Theorm (suite)

Dit autrement (pour simplifier) et exprimer la complexité uniquement par le big-O :

Supposons la relation de récurrence suivante :

$$T(n) = a T\left(\frac{n}{b}\right) + O(n^d)$$

Le "master theorem" permet d'obtenir une expression de la complexité de $T(n)$ comme suit :

- 1- Si $d < \log_b a$ alors $T(n) = O(n^{\log_b a})$.
 - Si (1) ne s'applique pas directement, on suppose $D(n) = n^d f(n)$ avec $d \geq 0$ et f est une fonction non décroissante (intuitivement f est une petite fonction comme $\log(n)$)
- 2- Si $d = \log_b a$ alors $T(n) = O(n^d \log(n))$.
- 3- Si $d > \log_b a$ alors $T(n) = O(n^d)$.

Complexité et Master Theorm (suite)

Exemples :

- Exemple 1 : $T(n) = 9T(n/3) + n$

Ici, $a = 9$, $b = 3$ d'où $n^{\log_b(a)} = n^{\log_3 9} = n^2$

De plus, $f(n) = n = O(n^{\log_3(9) - \epsilon})$ avec $\epsilon = 1$

→ on peut appliquer le cas 1 du MT : $T(n) = \theta(n^2)$

- Exemple 2 : $T(n) = T(2n/3) + 1$

Ici, $a = 1$, $b = 3/2$ d'où $n^{\log_b(a)} = n^{\log_{3/2}(1)} = n^0 = 1$

De plus, $f(n) = 1 = \theta(n^{\log_b(a)})$

→ on peut appliquer le cas 2 du MT : $T(n) = \theta(\log(n))$

- Exemple 3 : $T(n) = 3T(n/4) + n \log(n)$

Ici, $a = 3$, $b = 4$ d'où $n^{\log_b(a)} = n^{\log_4(3)} = O(n^{0.793})$

De plus, $f(n) = n \cdot \log(n) = \Omega(n) = \Omega(n^{\log_b(a) + \epsilon})$ avec $\epsilon \sim 0,2$

→ vérifions la condition de régularité :

pour n suffisamment grand, $af(n/b) = 3(n/4)\log(n/4) \leq 3/4 n \log(n) = cf(n)$ avec $c = 3/4$

→ on peut appliquer le cas 3 du MT : $T(n) = \theta(n \log(n))$

Exercices Master Theorem

- Appliquez le Master Theorem aux récurrences suivantes :

(a) $T(n) = 8T(n/2) + 1000n^2$

(b) $T(n) = 2T(n/2) + 10n$

(c) $T(n) = 8T(n/2) + n^2$

- Vérifier les complexités suivantes :

◦ $T(n) = 8T(n/2) + n^2 \leftrightarrow T(n) = \theta(n^3)$

◦ $T(n) = 2T(n/2) + n^2 \leftrightarrow T(n) = \theta(n^2)$

◦ $T(n) = 4T(n/2) + n^2 \leftrightarrow T(n) = \theta(n^2 \log(n))$

◦ Quels changements si la partie "driver" (non homogène) est de la forme $n^2 \log(n)$?

- Appliquez le Master Theorem aux récurrences suivantes :

◦ $T(n) = 8T(n/2) + n^3$

◦ $T(n) = 2T(n/2) + n^2$

◦ $T(n) = 9T(n/3) + n$

◦ $T(n) = 2T(n-2) + n^3$

Exercices Master Theorem (suite)

- Expliquez pourquoi on ne peut pas appliquer le Master Theorem aux récurrences suivantes :

(a) $T(n) = 2^n T(n/2) + n$

(b) $T(n) = 1/2 T(n/2) + n$

(c) $T(n) = 64T(n/8) - n^2 \log(n)$

(d) $T(n) = T(n/2) + n(2 - \cos(n))$

(e) $T(n) = 2T(n/2) + n/\log(n)$

- En effectuant le changement de variable $m = \log(n)$, résolvez la récurrence $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log(n)$

/

Exercice Stratégie

- Écrire un algorithme de complexité $\theta(n \cdot \log(n))$ qui détermine pour un ensemble de n entiers S et une valeur x s'il existe deux éléments dans S dont la somme est égale à x .
- Écrire une fonction *maxliste* qui reçoit une liste en paramètre, et qui renvoie le plus grand élément de cette liste en utilisant la technique **Diviser pour régner**.
- Écrire un algorithme de recherche de doublon dans une liste non triée de taille n puis calculer sa complexité.
 - La même chose pour une liste triée.
- ☞ N.B. : on peut envisager ou non des structures de données qui aideront à résoudre ces problèmes.

Exercice Stratégie (suite)

- Ecrire un algorithme qui produit toutes les permutations d'une liste.

→ P. Ex. , les permutations de $L=[1,2,5]$ seront :

$[[1, 2, 5], [1, 5, 2], [2, 1, 5], [2, 5, 1], [5, 1, 2], [5, 2, 1]]$

Calculer sa complexité de votre algorithme.

- ☞ Vérifiez vos calculs en utilisant la séquence suivante :

```
# Vérification
import itertools
lst=[1,2,5]
print(list(itertools.permutations(lst)))
```

Exercice Stratégie (suite)

- Soient deux chaînes de caractères (2 phrases) données dans deux tableaux T1 et T2 de taille N1 et N2 (on vérifiera si les 2 tailles sont identiques).
 - Vérifier si on a une anagramme (mots différents utilisant les mêmes lettres).

Solution à développer (+ algorithmes) :

- Traitement commune à toutes les solutions proposées :
 - Supprimer les espaces dans chaque phrase : $O(N)$ avec $N = \max(N1, N2)$.
 - Si les deux tableaux restants n'ont pas la même taille N alors échec
 - Transformer tous les caractères de T1 et de T2 en minuscule (ou en majuscule) (complexité $O(N)$)
 - Complexité de la partie commune?: $4.O(N) = O(N)$

Exercice Stratégie (suite)

- Soit deux tableaux T1 et T2 d'entiers de taille N dont les éléments sont y

cas 1 : éléments uniques

→ $O(N \log(N))$ si tri+comparaison, $O(N^2)$ si pas trié

cas 2 : éléments non uniques → Idem

- Vérifier que les deux tableaux contiennent les mêmes éléments.
- Une version pour le cas des éléments uniques est en annexes.
- Étudier la complexité de l'algorithme palindrome dans ses différentes versions.
 - Itératif, récursif, à l'aide de l'inversion, ...

Accepteur simple

- Une machine de Turing reçoit des chaînes MOT de 0 et de 1 en entrée. Elle renvoie True si MOT vérifie $L = \{0^k 1^k | k \geq 0\}$, False sinon.
 - Écrire un algorithme de complexité $O(n^2)$ qui réalise cette tâche
 - Proposer un algorithme de complexité $O(n \log(n))$ qui réalise la même tâche

Exercice Pesage

- On veut peser des quantités de farine de 1 à 40 Kg.

Quel serait le plus petit nombre de poids qui peuvent être utilisés sur une balance pour gérer l'une de ces quantités ?

- La première idée serait d'utiliser des poids de 1, 2, 4, 8, 16 et 32 kg.

C'est un nombre minimal si nous nous limitons à mettre des poids d'un côté et la farine de l'autre.

(I) Ecrire la fonction **pesage10** qui permet de montrer cela.

- Mais il est possible de mettre des poids sur les deux plateaux de la balance (gauche et droit).

- Dans ce cas, nous n'aurons besoin que de quatre poids: 1, 3, 9, 27 kg.

(II) Ecrire la fonction **pesage2(kg)** qui calcule la répartition des poids sur les plateaux permettant de peser n'importe quel poids de 1 à 40. Voir Remarques page suivante.

- Donner la complexité de votre algorithme.

- Dans les deux algorithmes ci-dessus, comme pour Fibonacci, nous recalculons beaucoup.

(III) Mettre en place le principe de "mémoïsation" dans votre solution et comparer les temps.

Exercice Pesage (suite)

Remarques :

- En fait, tout entier entre -40 et 40 peut être décomposé (ré-écrit) comme une combinaison linéaire de 1, 3, 9, 27 avec des scalaires pris dans l'ensemble $\{-1, 0, 1\}$.

Par exemple (les valeurs entre parenthèses sont celles prises dans $\{-1, 0, 1\}$):

$$7 = (1) * 1 + (-1) * 3 + (1) * 9 + (0) * 27$$

Ou

$$35 = (-1) * 1 + (0) * 3 + (1) * 9 + (1) * 27$$

- Pour **pesage20**, par convention, $(-1) * p$ place le poids p en plateau gauche, $(+1) * p$ place le poids p en plateau droit et $(0) * p$ veut dire : le poids p n'est pas utilisé.
- ☞ Il est évident que pour les valeurs négatives ($[-40 .. -1]$), le négation vient d'inverser les plateaux : si vous savez donner la solution pour 1..40Kg, en inversant les plateaux (*gauche* \leftrightarrow *droit*), vous obtiendrez la solution pour -40 .. -1 !

K premiers

- Proposer un algorithme qui permet de donner les k premiers dans une liste de scores de taille n .
- ☞ Plusieurs approches sont possibles :
 - On trouve le maximum, on le retire de la liste et on recommence : $O(k.n)$
 - On trie les scores et on retient les k plus grands : $O(n.\log(n)) (+k)$

Variant 1

- Variant 1 : Proposer un algorithme pour résoudre le problème de **la médiane des médianes** : il s'agit de trouver le k ème élément le plus grand au sein d'une liste non triée.
 - On a vu (en cours) une solution en cours en s'inspirant de l'algorithme "quick select" de *Hoar* avec une complexité ($O(n)$).

Indications : Ici, on veut suivre un autre principe qui se déroule en 3 étapes :

- On divise la liste en groupes de cinq éléments. Ensuite, pour chaque groupe de cinq, la médiane est calculée (effectuée en temps constant, même en utilisant un algorithme de tri).
- L'algorithme est alors appelé récursivement sur cette sous-liste de $n/5$ éléments pour trouver la vraie médiane de ces éléments.
On peut alors garantir que l'élément obtenu se place entre le 30e et le 70e centile.
- Enfin, la médiane des médianes est choisie pour être le pivot. Selon la position de l'élément recherché, l'algorithme recommence avec les éléments au-dessus du pivot ou en dessous, qui représentent au plus 70% de la taille initiale de l'espace de recherche.

Variant 2

- Variant 2 : trouvez le k ème élément dans un tableau où ce k ème élément est supérieur à k éléments du tableau.

Autrement dit, le 0-ème plus grand est le plus petit élément, le 3ème plus grand est supérieur à trois éléments (aurait l'indice 3 si le vecteur est trié) et ainsi de suite.

On suppose qu'il n'y ait pas d'éléments en double dans le tableau..

Votre solution utilisera une fonction de partition tiré p. ex. de *Quicksort*.

Cette fonction appelée *partition* s'exécute en $O(n)$ pour un tableau de n éléments.

Pour un tableau de n éléments, l'appel $FindKth(a, 0, n - 1, k)$ renvoie le k ème élément du tableau.

☞ Cet élément sera à l'indice de k si le tableau était trié.

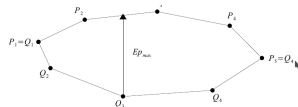
Discuter de la complexité de votre solution.

Exercice Épaisseur

- Un robot muni d'une pince manipule des objets.
→ Pour simplifier, on se place dans une dimension 2.

Chaque objet est représenté par un convexe bidimensionnel comme la figure ci-contre :

On étudie donc l'épaisseur verticale (et vu du dessus) d'un objet.



Chaque objet est donné par un polygone convexe représenté par deux liste de points (sommets) : les points "supérieurs" (de taille n) et "inférieurs" (m).

Pour simplifier, on suppose que ces 2 listes ont pour même point de départ : le sommet le plus à gauche. De même pour le sommet de fin le plus à droite.

Un point est composé de ses coordonnées x, y ; un segment est composé de deux points consécutifs.

Exercice Épaisseur (suite)

L'épaisseur maximale est forcément la perpendiculaire depuis un sommet sur une bordure sur un segment de la bordure opposée.

P. Ex. : depuis un sommet $P[i]$ du côté supérieur sur un segment du côté inférieur $(Q[j], Q[j + 1])$.

Une fonction de calcul de la distance entre un sommet et un segment doit être fournie (*calcDist(point, segment)*).

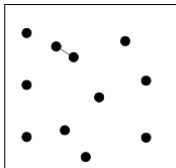
- Donner un algorithme qui calcule une paire *point* \times *segment* donnant l'épaisseur maximale d'un objet où le point est sur la bordure opposée de celle du segment.
- Calculer sa complexité.
- ☞ On évitera les cas symétriques (si vous en rencontrez !)
- ☞ Il existe des solutions de complexités diverses.

Exercice : Deux plus proches points

- Un problème de géométrie classique visant à trouver la paire de points la plus proche (closest pair) dans un plan.

Soit n points $((x_i, y_i))$ dans un plan.

- Trouver la paire de points les plus proches parmi ces points.



- ☞ Ce problème est étudié en cours et un algorithme est proposé.

Fibonacci et sa complexité

- La suite de Fibonacci est définie par :

$$F(0) = 1, F(1) = 1 \quad \text{et} \quad \forall n \geq 2, F(n) = F(n-1) + F(n-2)$$

- (I) Ecrire une fonction récursive *fibonacci* basée sur cette définition.

Ajoutez-y la mesure du temps à l'aide de *time.time()*.

- (II) Donnez le temps pour calculer Fib(2), Fib(4), Fib(8) et Fib(16).

- (III) Donnez la complexité de cet algorithme.

→ Correspond-t-elle aux temps mesurés ci-dessus (empiriquement)?

- On constate que cette définition récursive est très inefficace (à cause des re-calculs).

Une technique appelée "mémoïsation" (aka. tabing) permet de mémoriser les résultats intermédiaire pour éviter de les refaire.

- (IV) Ecrire une version de Fibonacci qui met cette technique en place.

→ Indication : utilisez un dico dont les éléments seront de la forme $n : F(n)$ permettant de mémoriser les couples n et $F(n)$

- (V) Quelle est la nouvelle complexité?

Python et complexité des listes

concernent tout le beginning (because everything after that must move), it gets more

Operation	Average Case	 Amortized Worst Case
Copy	$O(n)$	$O(n)$
Append[1]	$O(1)$	$O(1)$
Pop last	$O(1)$	$O(1)$
Pop intermediate[2]	$O(n)$	$O(n)$
Insert	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(1)$
Set Item	$O(1)$	$O(1)$
Delete Item	$O(n)$	$O(n)$
Iteration	$O(n)$	$O(n)$
Get Slice	$O(k)$	$O(k)$
Del Slice	$O(n)$	$O(n)$
Set Slice	$O(k+n)$	$O(k+n)$
Extend[1]	$O(k)$	$O(k)$
 Sort	$O(n \log n)$	$O(n \log n)$
Multiply	$O(nk)$	$O(nk)$
x in s	$O(n)$	
min(s), max(s)	$O(n)$	
Get Length	$O(1)$	$O(1)$

TabMat

- 1 Introduction
- 2 Exemples de complexité d'algos
- 3 Calcul de complexité
 - Calculs : quelques règles simples
- 4 Calculs : quelques équivalences
 - Comparaison symbolique de complexité
- 5 Exercices : Vérification de complexité
 - Vérifiez des complexités
- 6 Exercices de complexité
 - Rappel du cours
 - Exercices de démonstration de complexité
 - Exercices combinaison complexités
- 7 Exercices sur les limites
 - Calculs : équation de récurrence
 - Calcul par arbre de récursivité
 - Calcul par substitution
 - Exercices arbre/subst
 - Exemples de calculs
 - Exercice
- 8 Exercice : induction

TabMat (suite)

- Induction : l'importance de cas de base

- 9 Résolution équation de récurrence
 - Exercices éq de récurrence homogènes
 - Exos équation de récurrence non homogènes

- 10 Complexité et le Master Theorm

- 11 Exercices Master Theorem

- 12 Exercice : Algo + calcul de complexité

- Exercice Stratégie
- Accepteur simple
- Pesage
- K premiers
- Variant 1
- Variant 2
- Épaisseur
- Deux plus proches points
- Fibonacci

- 13 Complexité des opérations Python sur les listes

- 14 Table des matières