Chap-2 : Complexité et optimalité

Année universitaire :2020/2021

1. Définition de la complexité

1.1 Définition (Complexité)

- *La complexité d'un algorithme est la mesure du nombre d'opérations fondamentales qu'il effectue sur un jeu de données. La complexité est fonction de la taille du jeu de données.*

- Nous notons *Dn* l'ensemble des données de taille *n* et *T*(*d*) le coût de l'algorithme sur la donnée *d*.

- On définit 3 types de complexité :
  - Complexité au meilleur
  - Complexité au pire
  - Complexité moyenne

1. Définition de la complexité

1.2 Complexité au meilleur

$$T_{\min}(n) = \min_{d \in D_n} C(d)$$

C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à *n*.

C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille *n*.

1. Définition de la complexité

1.3 Complexité au pire

$$T_{\max}(n) = \max_{d \in D_n} C(d)$$

- C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à *n*.

- **Avantage :** il s'agit d'un maximum, et l'algorithme finira donc toujours avant d'avoir effectué *T*max(*n*) opérations.

- **Inconvénient :** cette complexité peut ne pas refléter le comportement «usuel» de l'algorithme. Le pire cas ne peut se produire que très rarement, mais il n'est pas rare que le cas moyen soit aussi mauvais que le cas pire.

1. Définition de la complexité

1.4 Complexité en moyenne

$$T_{moy}(n) = \frac{\sum_{d \in D_n} C(d)}{|D_n|}$$

- C'est la moyenne des complexités de l'algorithme sur des jeux de données de taille *n* (en toute rigueur, il faut bien évidemment tenir compte de la probabilité d'apparition de chacun des jeux de données).

- **Avantage :** reflète le comportement « général » de l'algorithme si les cas extrêmes sont rares ou si la complexité varie peu en fonction des données.

- **Inconvénient :** en pratique, la complexité sur un jeu de données particulier peut être nettement plus importante que la complexité en moyenne, dans ce cas la complexité en moyenne ne donnera pas une bonne indication du comportement de l'algorithme.

- En pratique, nous ne nous intéresserons qu'à la complexité au pire et à la complexité en moyenne

2. Définition de l'optimalité

- *Un algorithme est dit **optimal** si sa complexité est la complexité minimale parmi les algorithmes de sa classe.*

- Nous nous intéresserons quasi exclusivement à la *complexité en temps* des algorithmes. Il est parfois intéressant de s'intéresser à d'autres ressources, comme la *complexité en espace* (taille de l'espace mémoire utilisé), la largeur de bande passante requise, etc.

3. Illustration : cas du tri par insertion

3.1 Problématique du tri

- **Données** : une séquence de *n* nombres, *a*1, ..., *an*.
- **Résultats** : une permutation a'$_i$ de la séquence d'entrée, telle que

$$a'_i \leq a'_{i+1} \ \forall i$$

3. Illustration : cas du tri par insertion

3.2 Principe du tri par insertion

- De manière répétée, on retire un nombre de la séquence d'entrée et on l'insère à la bonne place dans la séquence des nombres déjà triés (ce principe est le même que celui utilisé pour trier une poignée de cartes).

3. Illustration : cas du tri par insertion

3.3 Algorithme

For *j*=2 **to** *n* **do**
clé = A[*j*]
i= *j*-1
while (*i* > 0 **and** A[*i*] > clé) **do**
A[*i*+1]=A[*i*]
i=*i*-1
End_while
A[*i*+1]=clé
Endfor
	On retire un nombre de la séquence d'entrée.
	Les j-1 premiers éléments de A sont déjà triés.
	Tant que l'on n'est pas arrivé au début du tableau, et que l'élément courant est plus grand que celui à insérer.
	On décale l'élément courant (on le met dans la place vide).
	Finalement, on a trouvé où insérer notre nombre.

3. Illustration : cas du tri par insertion

3.4 Complexité

- Attribuons un coût en temps à chaque instruction, et comptons le nombre d'exécutions de chacune des instructions. Pour chaque valeur de , nous notons t_j le nombre d'exécutions de la boucle « while » pour cette valeur de j. Il est à noter que la valeur de t_j **dépend des données**...

Instruction	Coût unitaire	Coût total
For j=2 **to** n **do**	$c1$	n
clé = A[j]	$c2$	$n-1$
i= j-1	$c3$	$n-1$
while (i > 0 **and** A[i] > clé) **do**	$c4$	$\sum_{j=2}^{n} t_j$
A[i+1]=A[i]	$c5$	$\sum_{j=2}^{n} (t_j - 1)$
i=i-1	$c6$	$\sum_{j=2}^{n} (t_j - 1)$
End_while		
A[i+1]=clé	$c7$	$n-1$
Endfor		

- Le temps d'exécution total de l'algorithme est alors la somme des coûts élémentaires:

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 (n-1)$$

3. Illustration : cas du tri par insertion

Complexité au meilleur

- le cas le plus favorable pour l'algorithme TRI-INSERTION est quand le tableau est déjà trié.

  Dans ce cas $t_j = 1$ pour tout j.

  $T(n) = c1n+c2(n-1)+c3(n-1)+c4(n-1)+c7(n-1)$

  $= (c1+c2+c3+c4+c7)n-(c2+c3+c4+c7)$:

  $T(n)$ peut ici être écrit sous la forme $T(n) = an+b$, a et b étant des *constantes* indépendantes des entrées, et

➔ **$T(n)$ est donc une fonction linéaire de n.**

- Le plus souvent, comme c'est le cas ici, le temps d'exécution d'un algorithme est fixé pour une entrée donnée; mais il existe des algorithmes « aléatoires » intéressants dont le comportement peut varier même pour une entrée fixée.

3. Illustration : cas du tri par insertion

Complexité au pire

- Le cas le plus défavorable pour l'algorithme TRI-INSERTION est quand le tableau est déjà trié dans l'ordre inverse. Dans ce cas $t_j = j$ pour tout j.
- Rappel :

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \quad donc \quad \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \quad et \quad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

$$T(n) = \frac{1}{2}(c_4 + c_5 + c_6)n^2 + \frac{1}{2}(2c_1 + 2c_2 + 2c_3 + c_4 - c_5 - c_6 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

- $T(n)$ est donc de la forme
- $T(n)$ est donc une **fonction quadratique** de n.

3. Illustration : cas du tri par insertion

Complexité en moyenne

- Supposons que l'on applique l'algorithme de tri par insertion à *n* nombres choisis au hasard. Quelle sera la valeur de *tj* ? C'est-à-dire, où devra-t-on insérer A[*j*] dans le sous-tableau A[1.. *j*-1] ?

- En moyenne, la moitié des éléments de A[1.. *j*-1] sont inférieurs à A[*j*], et l'autre moitié sont supérieurs. Donc *t j* = *j*/2. Si l'on reporte cette valeur dans l'équation définissant *T*(*n*), on obtient, comme dans le cas pire, une fonction quadratique en *n*.

3. Illustration : cas du tri par insertion

Ordre de grandeur

- Ce qui nous intéresse vraiment, c'est l'ordre de grandeur du temps d'exécution. Seul le terme dominant de la formule exprimant la complexité nous importe, les termes d'ordres inférieurs n'étant pas significatifs quand *n* devient grand. On ignore également le coefficient multiplicateur constant du terme dominant. On écrira donc, à propos de la complexité du tri par insertion :
- Complexité au meilleur = $O(n)$.
- Complexité au pire = $O(n^2)$.
- Complexité en moyenne = $O(n^2)$.

- En général, on considère qu'un algorithme est plus efficace qu'un autre si sa complexité dans le cas pire a un ordre de grandeur inférieur.

3. Illustration : cas du tri par insertion

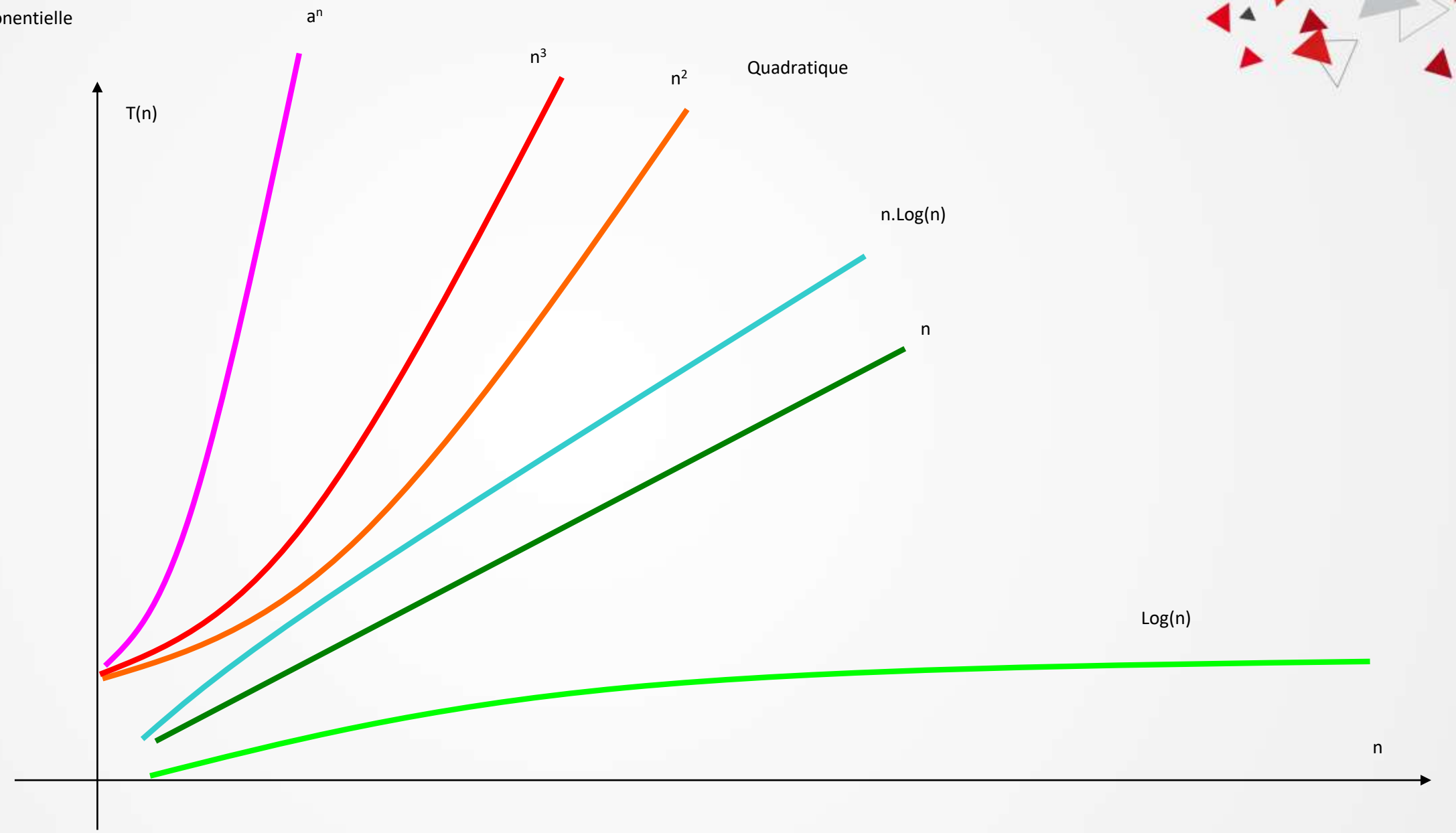
Classes de complexité

- Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité :

- ***O*(log*n*)** : Les algorithmes sub-linéaires dont la complexité est en général en $O(\log n)$.
- **O(n)** : Les algorithmes linéaires en complexité $O(n)$
- **O(nlogn)** : et ceux en complexité en $O(n\log n)$
- ***O*(n^k)** : Les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$
- **Exp(n)** : Les algorithmes exponentiels

- Les trois premières classes sont considérées rapides alors que la quatrième est considérée lente et la cinquième classe est considérée impraticable.

	$\log_2(n)$	n	$n\log_2(n)$	n^2	n^3	n^4	exp(n)
1	0,00	1	0,00	1	1	1	3
2	1,00	2	2,00	4	8	16	7
3	1,58	3	4,75	9	27	81	20
4	2,00	4	8,00	16	64	256	55
5	2,32	5	11,61	25	125	625	148
6	2,58	6	15,51	36	216	1296	403
7	2,81	7	19,65	49	343	2401	1097
8	3,00	8	24,00	64	512	4096	2981
9	3,17	9	28,53	81	729	6561	8103
10	3,32	10	33,22	100	1000	10000	22026
11	3,46	11	38,05	121	1331	14641	59874
12	3,58	12	43,02	144	1728	20736	162755
13	3,70	13	48,11	169	2197	28561	442413
14	3,81	14	53,30	196	2744	38416	1202604
15	3,91	15	58,60	225	3375	50625	3269017
16	4,00	16	64,00	256	4096	65536	8886111
17	4,09	17	69,49	289	4913	83521	24154953
18	4,17	18	75,06	324	5832	104976	65659969
19	4,25	19	80,71	361	6859	130321	178482301
20	4,32	20	86,44	400	8000	160000	485165195


Exponentielle
a^n
n^3
n^2
Quadratique
T(n)
n.Log(n)
n
Log(n)
n