Techniques mathématiques pour l'informatique

R. Watrigant

1 Analyse d'algorithmes

1.1 Comment évaluer l'efficacité des algorithmes ?

1.1.1 Introduction

Un problème peut être résolu par plusieurs algorithmes (ex : plusieurs façons de trier un tableau d'entiers), et par encore plus de programmes (ex : plusieurs langages de programmation). Question : comment comparer plusieurs algorithmes réalisant la même tâche ?
Plusieurs caractéristiques d'un algorithme :

- place mémoire nécessaire (quantifiable)
- durée d'exécution (quantifiable)
- simplicité du code (inquantifiable car subjectif...)

Exemple : l'algorithme A implémenté dans le langage P sur l'ordinateur O, et exécuté sur la donnée D utilise k secondes et j bits de mémoire. Que se passe-t-il si l'on exécute sur la donnée D' ? Si l'on change d'ordinateur ? $\Rightarrow$ On veut un cadre formel nous permettant d'affirmer "l'algorithme A est meilleur que l'algorithme B". Dans ce cours, on va mesurer seulement la durée d'exécution.

1.1.2 Cadre d'étude

Le temps d'exécution d'un algorithme dépend de l'entrée (le tri de 1000 nombres prend plus de temps que le tri de 3 nombres).
On exprimera le temps d'exécution d'un algorithme en fonction de la taille de son entrée (Attention, un algorithme peut demander des temps différents pour deux entrées de même taille).

Taille de l'entrée = nombre d'éléments constituant l'entrée. Exemples :

- pour un tableau : son nombre d'éléments
- pour un nombre : son nombre de bits nécessaires à sa représentation ($\log_2 n$ généralement)
- pour un graphe : son nombre de sommets

Temps d'exécution Axiome : *le temps d'exécution d'un algorithme est proportionnel au nombre d'étapes élémentaires.* Exemples :

- affectations
- opérations arithmétiques (additions, multiplications...)
- comparer deux nombres

Dans ce cours, une ligne de code = une étape élémentaire (Attention, on pourrait avoir dans une ligne de code l'instruction "trier le tableau T" qui ne serait pas une opération élémentaire).

1.1.3 Exemple : le tri par insertion

TRI_INSERTION

Input: : A : tableau de n entiers

1: **for** $j = 2$ à $longueur(A)$ **do**
2: $cle \leftarrow A[j]$
3: $i \leftarrow j - 1$
4: **while** $i > 0$ et $A[i] > cle$ **do**
5: $A[i+1] \leftarrow A[i]$
6: $i \leftarrow i - 1$
7: **end while**
8: $A[i+1] \leftarrow cle$
9: **end for**

Nombre d'exécutions de chaque instruction (remarque : pour les boucles For et $While$, le test est exécuté une fois de plus que le corps de la boucle) :

- instruction 1 : n
- instruction 2 : $n-1$
- instruction 3 :$n-1$
- instruction 4 : $\sum_{j=2}^{n} t_j$
- instruction 5 : $\sum_{j=2}^{n} t_j - 1$
- instruction 6 : $\sum_{j=2}^{n} t_j - 1$
- instruction 8 : $n-1$

Avec t_j le nombre de fois que le test de la boucle $While$ est exécuté pour cette valeur de j.
Soi $T(n)$ le temps d'exécution de TRI_INSERTION sur une entrée de taille n. On a :

$$\begin{aligned} T(n) &= n + (n-1) + (n-1) + (\sum_{j=2}^{n} t_j) + (\sum_{j=2}^{n} t_j - 1) + (\sum_{j=2}^{n} t_j - 1) + (n-1) \\ &= 2n - 1 + 3\sum_{j=2}^{n} t_j \end{aligned}$$

Cas favorable : si le tableau est déjà trié, : $t_j = 1$ pour tout $j = 2, ..., n$, et donc :

$$T(n) = 5n - 4$$

Cas défavorable : si le tableau est trié à l'envers : $t_j = j$ pour tout $j = 2, ..., n$, et donc :

$$\begin{aligned} T(n) &= 2n - 1 + 3\frac{n^2 + n - 2}{2} && (1) \\ &= \frac{3}{2}n^2 + \frac{7}{2}n - 4 && (2) \end{aligned}$$

$\Rightarrow$ fonction quadratique de n. Sauf mention du contraire, on regardera toujours les performances d'un algorithme dans le pire des case. Plusieurs raisons :

- permet de borner supérieurement le temps pour toute les données de taille n
- ce cas peut apparaitre fréquemment en pratique
- cas "moyen" presque aussi mauvais que le cas le plus défavorable (dans notre exemple : prendre $t_j = j/2$ donne une fonction quadratique)

1.2 Croissance des fonctions, notations de Landau

Pour le tri par insertion, on a vu que dans le cas le plus défavorable, $T(n) = \frac{3}{2}n^2 + \frac{7}{2}n - 4$. Pour les entrées suffisamment grandes, seul l'ordre de grandeur du temps d'exécution compte : on "enlève" les constantes multiplicatives et les termes d'ordres inférieurs. On notera $T(n) = \theta(n^2)$, et on dira que l'algorithme du tri par insertion a une complexité en $\theta(n^2)$.
Dans ce qui suit, toutes les fonctions vont de $\mathbb{N}$ dans $\mathbb{N}$.
Abus de notation : au lieu de "$f : n \mapsto f(n)$", on notera directement "$f(n)$"

Notation θ : Pour une fonction donnée $g(n)$, on note $\theta(g(n))$ ("theta de g de n") l'ensemble de fonctions suivant :

$$\theta(g(n)) = \{f(n) : \exists c_1, c_2 > 0 \text{ et } n_0 \in \mathbb{N} \text{ tels que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ pour tout } n \geq n_0\}$$

Si $f(n) \in \theta(g(n))$, on dira que $g(n)$ est une borne asymptotiquement approchée de $f(n)$.
Abus de notation : on notera aussi $f(n) = \theta(g(n))$
Exemple : montrons que $\frac{1}{2}n^2 - 3n = \theta(n^2)$. Montrons qu'on peut trouver c_1, c_2 et n_0 tels que $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ pour tout $n \geq n_0$. En divisant par n^2, on a

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

On remarque que $c_2 \geq 1/2$ convient pour tout $n \geq 1$, et $c_1 \geq 1/14$ convient pour tout $n \geq 7$.
Exercice : montrer que $6n^3 \neq \theta(n^2)$ et que $20n \neq \theta(n^2)$.

Notation O : Pour une fonction donnée $g(n)$, on note $O(g(n))$ ("grand O de g de n") l'ensemble de fonctions suivant :

$$O(g(n)) = \{f(n) : \exists c > 0 \text{ et } n_0 \in \mathbb{N} \text{ tels que } 0 \leq f(n) \leq cg(n) \text{ pour tout } n \geq n_0\}$$

Si $f(n) \in O(g(n)$, on dira que $g(n)$ est une borne supérieure asymptotique de $f(n)$. Abus de notation : on notera aussi $f(n) = O(g(n))$.
Propriété : $\theta(g(n)) \subseteq O(g(n))$.
Exercice : montrer que $20n = O(n^2)$.

Notation Ω : Pour une fonction donnée $g(n)$, on note $\Omega(g(n))$ ("grand oméga de g de n") l'ensemble de fonctions suivant :

$$\Omega(g(n)) = \{f(n) : \exists c > 0 \text{ et } n_0 \in \mathbb{N} \text{ tels que } 0 \leq cg(n) \leq f(n) \text{ pour tout } n \geq n_0\}$$

Si $f(n) \in \Omega(g(n)$, on dira que $g(n)$ est une borne inférieure asymptotique de $f(n)$. Abus de notation : on notera aussi $f(n) = \Omega(g(n))$.
Propriété : $\theta(g(n)) \subseteq \Omega(g(n))$ Exercice : montrer que $5n^3 + 2n = \Omega(n^2)$

Théorème 1. *Pour deux fonctions quelconques f et g, on a $f(n) = \theta(g(n))$ si et seulement si $f(n) = O(g(n))$ et $f(n) = \Omega(g(n)$.*

Démonstration. On a déjà $\theta(g(n)) \subseteq \Omega(g(n))$ et $\theta(g(n)) \subseteq O(g(n))$, il ne reste donc plus qu'à montrer que $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$ implique $f(n) = \theta(g(n)$. Il existe c_1 et n_0^1 tels que $f(n) \leq c_1 g(n)$ pour tout $n \geq n_0^1$, et il existe c_2 etn_0^2 tels que $f(n) \geq c_2 g(n)$ pour tout $n \geq n_0^2$. D'où, pour tout $n \geq \max\{n_0^1, n_0^2\}$, on a $c_2 g(n) \leq f(n) \leq c_1 g(n)$ et donc $f(n) = \theta(g(n))$. □

Notation o : Pour une fonction donnée $g(n)$, on note $o(g(n))$ ("petit o de g de n") l'ensemble de fonctions suivant :

$$o(g(n)) = \{f(n) : \forall c > 0 \exists n_0 \in \mathbb{N} \text{ tels que } 0 \leq f(n) < cg(n) \text{ pour tout } n \geq n_0\}$$

Propriété : $f(n) = o(g(n))$ équivaut à $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
Exercice : montrer que $56n = o(n^2)$ et que $\frac{1}{2}n^2 \neq o(n^2)$

Notation ω : Pour une fonction donnée $g(n)$, on note $\omega(g(n))$ ("petit oméga de g de n") l'ensemble de fonctions suivant :

$$\omega(g(n)) = \{f(n) : \forall c > 0 \exists n_0 \in \mathbb{N} \text{ tels que } 0 \leq cg(n) < f(n) \text{ pour tout } n \geq n_0\}$$

Propriété : $f(n) = \omega(g(n))$ équivaut à $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.
Exercice : montrer que $\frac{1}{2}n^2 = \omega(n)$ et que $\frac{1}{2}n^2 \neq \omega(n^2)$

Analogie avec la comparaison de nombres

$$\begin{aligned} f(n) = O(g(n)) &\approx a \leq b \\ f(n) = o(n)) &\approx a < b \\ f(n) = \theta(g(n)) &\approx a = b \\ f(n) = \Omega(g(n)) &\approx a \geq b \\ f(n) = \omega(g(n)) &\approx a > b \end{aligned}$$

Ordres de grandeur classiques

- $O(1)$: constant
- $O(\log(n)$: logarithmique
- $O(n)$: linéaire
- $O(n\log(n))$: $n\log(n)$ (ou quasi-linéaire)
- $O(n^2)$: quadratique
- $O(n^3)$: cubique
- $O(n^k), k > 0$: polynomial
- $O(2^n)$: exponentiel

1.3 Récurrences

Comment évaluer la complexité des algorithmes récursifs ?

1.3.1 Premier exemple : factorielle

On rappelle l'algorithme récursif du calcul de la factorielle d'un entier :

FACTORIELLE(n)
calcule et retourne $n!$

```
1: if n = 0 then
2:    return 1
3: else
4:    return n * factorielle(n − 1)
5: end if
```

Observons la trace de l'algorithme pour $n = 4$:

↪ appel de $factorielle(4)$: on retourne $4 * factorielle(3)$
  ↪ appel de $factorielle(3)$: on retourne $3 * factorielle(2)$
    ↪ appel de $factorielle(2)$: on retourne $2 * factorielle(1)$
      ↪ appel de $factorielle(1)$: on retourne $1 * factorielle(0)$
        ↪ appel de $factorielle(0)$: on retourne 1

Ici, on voit bien que le nombre d'instructions est en $\theta(n)$ (attention : ce n'est pas un algorithme polynômial, car (1) la taille de l'entrée est $\log n$, et (2) on doit calculer $n!$). Possibilité de dérécurser l'algorithme :

FACTORIELLE(n)
calcule et retourne $n!$

1: $fact \leftarrow 1$
2: **for** $i \leftarrow 1$ à n **do**
3: $\quad fact \leftarrow fact * i$
4: **end for**
5: **return** $fact$

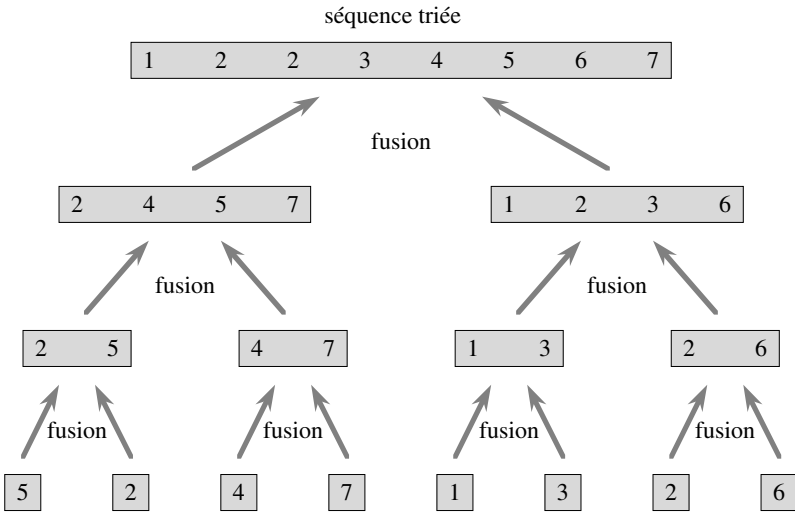
Ici, la version itérative est également en $\theta(n)$.

1.3.2 Second exemple : tri par fusion

Idée : "diviser-pour-régner" :
- **Diviser** : Diviser la suite de n éléments à trier en deux sous-suites de $n/2$ éléments chacune.
- **Régner** : Trier les deux sous-suites de manière récursive en utilisant le tri par fusion.
- **Combiner** : Fusionner les deux sous-suites triées pour produire la réponse triée.

Cas de base : si la séquence à trier est de longueur 1, alors elle est déjà triée.

Exemple d'exécution



FUSION(A, p, q, r)

Fusionne les listes $A[p...q]$ et $A[(q+1)...r]$ qui sont supposées être triées et met le résultat dans $A[p...r]$

Input: : A, p, q, r

```
1: n_1 ← p − q + 1
2: n_2 ← r − q
3: créer deux tableaux L[1...(n_1 + 1)] et R[1...(n_2 + 1)] //+1 pour les sentinelles
4: for i ← 1 à n_1 do
5:     L[i] ← A[p + i − 1]
6: end for
7: for j ← 1 à n_2 do
8:     R[j] ← A[q + j]
9: end for
10: L[n_1 + 1] ← ∞ //on place des sentinelles
11: R[n_2 + 1] ← ∞ //on place des sentinelles
12: i ← 1
13: j ← 1
14: for k ← p à r do
15:     if L[i] ≤ R[j] then
16:         A[k] ← L[i]
17:         i ← i + 1
18:     else
19:         A[k] ← L[i]
20:         j ← j + 1
21:     end if
22: end for
```

TRI_FUSION(A, p, r)

Trie les éléments de p à r d'un tableau A

Input: : A, p, r

```
1: if p < r then
2:     q ← ⌊(p + r)/2⌋
3:     TRI_FUSION(A, p, q)
4:     TRI_FUSION(A, q + 1, r)
5:     FUSION(A, p, q, r)
6: end if
```

Analyse de la complexité de l'algorithme

Soit $D(n)$ le temps pour diviser le problème en sous-problèmes, $C(n)$ pour construire la solution finale à partir des solutions aux sous-problèmes, et $T(n)$ le temps de l'algorithme global.

- **Diviser** : calcule simplement le milieu du sous-tableau : $D(n) = \theta(1)$.
- **Régner** : on résoud deux sous-problèmes, chacun ayant une entrée de taille $n/2$, ce qui contribue à $2T(n/2)$ au temps d'exécution.
- **Combiner** : la procédure FUSION prend un temps $\theta(n)$, donc $C(n) = \theta(n)$.

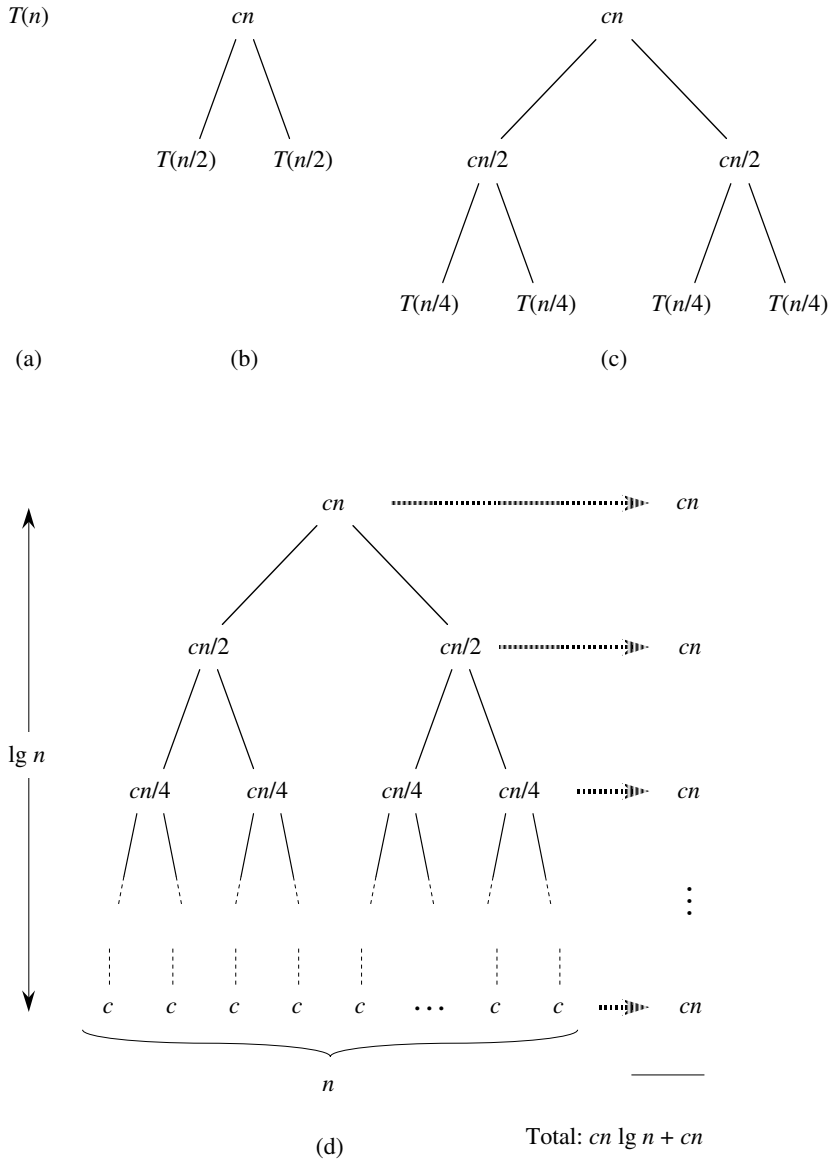
On a donc :

$$T(n) = \begin{cases} \theta(1) & \text{si} \quad n = 1 \\ 2T(n/2) + \theta(n) & \text{si} \quad n > 1 \end{cases} \tag{3}$$

ou :

$$T(n) = \begin{cases} c & \text{si} \quad n = 1 \\ 2T(n/2) + cn & \text{si} \quad n > 1 \end{cases} \tag{4}$$

(ce n'est pas le même c mais on peut contourner en prenant le max des deux).
Pour résoudre la récurrence, on peut utiliser un arbre de récursivité (on suppose que n est une puissance de 2) :



D'où l'algorithme FUSION a une complexité en $\theta(n \log(n))$ (mieux que TRI_INSERTION).

1.3.3 Troisième exemple : les tours de Hanoï

Le problème des tours de Hanoï est le suivant : étant donnés n disques de tailles différentes empillés du plus grand (en bas) au plus petit (en haut) sur une tour de départ D, déplacer ces n disques vers une tour d'arrivée A en utilisant une tour intermédiaire I en respectant les règles suivantes :

- ne déplacer qu'un disque à la fois
- empiller un disque sur un disque plus grand uniquement (ou sur une tour vide)

Algorithme de résolution :

HANOI(n, D, A, I)
déplace les n premiers disques de la tour D à la tour A en utilisant la tour I

1: **if** $n = 1$ **then**
2: déplacer le disque de D à A
3: **else**
4: Hanoi($n-1$, D, I, A)
5: déplacer le disque restant de D vers A
6: Hanoi($n-1$, I, A, D)
7: **end if**

Soit $T(n)$ le nombre d'instructions (déplacements) appelées. On a $T(n) = 2T(n-1)+1$.

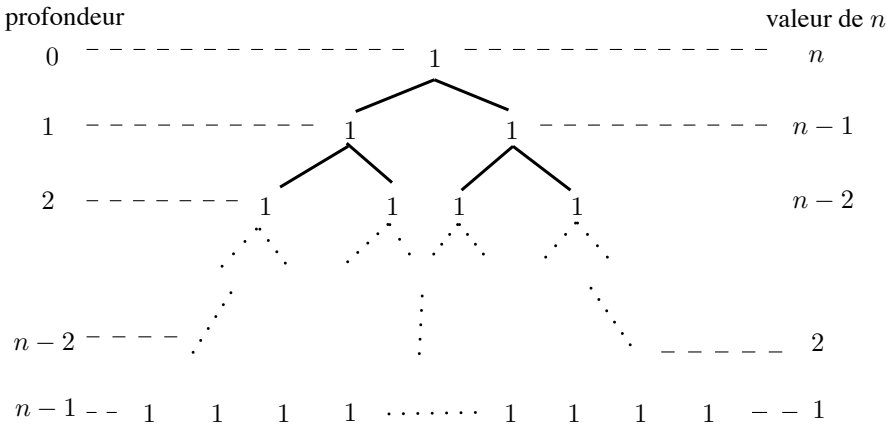
Première méthode : méthode empirique + vérification : calculons les premières valeurs :

$$T(1) = 1 T(2) = 3 T(3) = 7 T(4) = 15 T(5) = 31$$

On conjecture $D(n) = 2^n - 1$. Vérifions :

$$\begin{aligned} D(n+1) &= 2D(n)+1 \\ &= 2(2^n-1)+1 \\ &= 2^{n+1}-2+1 \\ &= 2^{n+1}-1 \end{aligned}$$

Seconde méthode : arbre d'appels : on construit l'arbre des appels



$D(n)$ est en fait le nombre de noeuds de l'arbre. Nombre de noeuds à la profondeur $i = 2^i$, d'où

$$D(n) = \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

1.3.4 Le cas général

Théorème 2. *(admis)*
Soit $a \geq 1$ et $b > 1$ deux constantes, f une fonction et T définie pour les entiers non négatifs par la récurrence

$$T(n) = aT(n/b) + f(n)$$

Où n/b peut également être interprété comme $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$.
$T(n)$ peut alors être bornée asymptotiquement de la façon suivante :

- *1) si $f(n) = O(n^{\log_b(a)-\epsilon})$ pour une certaine constante $\epsilon > 0$, alors $T(n) = \theta(n^{\log_b(a)})$.*
- *2) si $f(n) = \theta(n^{\log_b(a)})$ alors $T(n) = \theta(n^{\log_b(a)} \log(n))$.*
- *3) si $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ pour une certaine constante $\epsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une certaine constante $c < 1$ et pour tout n suffisamment grand, alors $T(n) = \theta(f(n))$.*

Exemple :

$$T(n) = 9T(n/3) + n$$

On a $a = 9$, $b = 3$ et $f(n) = n$. On a $n^{\log_b(a)} = n^{\log_3(9)} = \theta(n^2)$. D'où $f(n) = O(n^{\log_3(9)-\epsilon})$ avec $\epsilon = 1$, on peut appliquer le cas 1 du théorème général, et donc $T(n) = \theta(n^2)$.

Attention : tous les cas ne sont pas traités par le théorème général :

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

On a bien $a = 2$, $b = 2$, $f(n) = n\log(n)$ et $n^{\log_b(a)} = n$. On pourrait penser que le cas 3 s'applique car $f(n)$ est asymptotiquement plus grande que $n^{\log_b(a)} = n$, mais elle n'est pas *polynomialement* plus grande : le rapport $\frac{f(n)}{n^{\log_b(a)}} = \frac{n\log(n)}{n} = \log(n)$ est asymptotiquement plus petit que n^ϵ pour toute constante strictement positive ϵ.

2 Introduction à la théorie de la complexité

2.1 Cadre d'étude

Dans le chapitre précédent, on a vu qu'il était possible de comparer l'efficacité d'algorithmes effectuant la même tâche. La théorie de la complexité vise à classer les problèmes combinatoires selon leur difficulté à pouvoir les résoudre. Qu'est-ce qu'un problème combinatoire ?

On distingue les **problèmes de décision** et les **problèmes d'optimisation**. Les problèmes de décisions retournent une réponse binaire (vrai ou faux), tandis que les problèmes d'optimisation retournent des objets mathématiques (généralement l'élément qui maximise ou minimise une fonction parmi un ensemble d'objets) .
Exemple de problème de décision :
Test de primalité
Entrée : un entier n
Question : l'entier n est-il premier ?

Exemple de problème d'optimisation :

> Sous tableau de somme maximale
> **Entrée :** un tableau T d'entiers
> **Réponse :** deux indices i et j
> **But :** maximiser la somme des éléments de T de i à j

A un problème d'optimisation correspond un problème de décision, exemple :

> Sous tableau de somme maximale (décision)
> **Entrée :** un tableau T d'entiers, un entier S
> **Question :** existe-t-il deux indices i et j tels que la somme des éléments de T de i à j vaut au moins S ?

Il est facile de voir que si l'on sait résoudre le problème d'optimisation, alors on sait résoudre le problème de décision. Réciproquement, s'il est difficile de résoudre le problème de décision, il sera difficile de résoudre le problème d'optimisation. C'est pour cela que l'on s'intéresse plus généralement aux problèmes de décision.

2.2 Les classes P et NP

2.2.1 La classe P

La théorie de la complexité vise à séparer les problèmes en **classes** (ensembles).

Définition 1. *La classe P (pour Polynomial) est la classe des problèmes de décision tels qu'il existe un algorithme polynômial pour les résoudre.*

Exemples de problèmes polynomiaux :

> Plus grand élément
> **Entrée :** un tableau T d'entiers
> **Question :** existe-t-il un élément de T strictement plus grand que tous les autres ?

Un algorithme naïf est de tester toutes les paires de sommets. Si T contient n éléments, cet algorithmes s'exécute en $O(n^2)$.

Sous tableau de somme maximale (décision)
Entrée : un tableau T d'entiers, un entier S
Question : existe-t-il deux indices i et j tels que la somme des éléments de T de i à j vaut au moins S ?

On a vu précédemment que l'on pouvait résoudre le problème d'optimisation associé en $O(n\log(n))$.

2.2.2 Certificats et classe NP

Définition 2. *Etant donné une instance I d'un problème Π. Un certificat C est un objet qui prouve l'existance d'une solution. Le certificat est polynomial si C est de taille polynomiale en la taille de I, et s'il existe un algorithme polynomial prenant en entrée I et$C$ et retourne vrai si et seulement si I est une instance positive de Π.*

Définition 3. *Un problème de décision Π est dans la classe NP (pour Non-deterministic Polynomial) s'il existe pour chaque solution positive un certificat polynomial.*

Exemple : pour le problème Sous tableau de somme maximale (décision), un certificat est un couple (i, j), il est bien de taille polynomiale en la taille du tableau, et on peut vérifier que la solution est positive en calculant la somme des élément de T de i à j, et comparant celle-ci avec l'entier S de l'instance.

Observation : on a clairement $P \subseteq NP$ car on peut tester en temps polynomial si une instance est positive ou non.
La question de savoir si $NP \subseteq P$ est ouverte.

2.3 Les problèmes NP-difficiles et NP-complets

Définition 4. *Soit Π_1 et Π_2 deux problèmes de décision. Une réduction polynomiale de Π_1 à Π_2 est un algorithme polynomial qui prend en entrée une instance de Π_1, et retourne une instance de Π_2 telle que l'instance retournée est positive pour Π_2 si et seulement si l'instance d'entrée est positive pour Π_1.*

S'il existe une réduction polynomiale de Π_1 à Π_2, on dit que Π_1 se réduit à Π_2.

Observation : Si Π_1 se réduit à Π_2, et qu'il existe un algorithme polynomial pour Π_2, alors il existe également un algorithme polynomial pour Π_1.

Définition 5. *Un problème Π_1 est NP-difficile si pour tout problème Π_2 de NP, il existe une réduction polynomiale de Π_2 vers Π_1*

Définition 6. *Un problème Π est NP-complet si :*
- $\Pi \in NP$
- Π *est NP-difficile*

Autrement dit, si Π_1 est NP-complet, alors un algorithme polynomial pour Π_1 implique un algorithme polynomial pour tous les problèmes de NP. D'où l'intérêt de la question $P = NP$.