Chapitre 1
Introduction à la complexité algorithmique

L'objectif de ce premier chapitre est d'introduire la notion de complexité algorithmique et de fournir les méthodes et les outils mathématiques fondamentales pour l'analyse de la complexité des algorithmes.

1- Notion de complexité algorithmique

Le but d'un algorithme est de proposer une solution informatique à un problème de calcul. Un algorithme est une procédure effective qui prend en entrée un ensemble de données et qui fournit en sortie un ensemble de résultats. La procédure effective de l'algorithme est décrite au moyen d'une séquence finie d'étapes de calculs qui vont transformer les données en résultats. Pour être exécutées, les algorithmes doivent être traduits en programmes écrits dans certains langages de programmation, comme C, C++ et Java.

L'efficacité ou la performance d'un algorithme dépend de sa capacité à minimiser certains critères très importants, comme :

- Le temps d'exécution.
- L'espace mémoire utilisé.
- La bande passante du réseau.

La complexité algorithmique est l'étude de la performance des algorithmes, notamment en terme de temps d'exécution. Cette notion peut paraître abstraite, cependant elle est bien concrète, et son incidence sur le fonctionnement d'un programme est réelle. Quelle est l'utilité d'un algorithme qui produit un programme qui donne le résultat au bout d'une semaine ou d'un mois ? Il est alors indispensable d'évaluer la performance d'un algorithme avant de le programmer.

Il est fréquent qu'un même problème possède plusieurs solutions décrites par des algorithmes différents. La comparaison entre ces algorithmes passe par l'étude de leur complexité.

La complexité algorithmique permet de s'informer sur l'efficacité intrinsèque d'un algorithme, c'est-à-dire sur la performance "naturelle" d'un algorithme en dehors de l'environnement dans lequel sera implémenté ce dernier : machine physique, système d'exploitation, compilateurs de programmes, …etc. Considérons le problème de tri dans l'ordre croissant d'un tableau de N éléments, et s'occupons nous du nombre de comparaisons entre les éléments du tableau. Trois algorithmes permettent de résoudre ce problème :

- L'algorithme de tri par sélection du minimum : effectue au maximum $c_1 N^2$ comparaisons.
- L'algorithme de tri par insertion : effectue au maximum $c_2 N^2$ comparaisons.

- L'algorithme de tri par fusion : effectue au maximum $c_3 N(\log_2 N)$ comparaisons.

Les constantes c_1 et c_2 sont plus petites par rapport à la constante c_3. Lorsqu'il s'agit de trier un tableau de taille gigantesque, et si l'on exécute l'algorithme de tri par fusion sur une machine plus lente, ayant un système d'exploitation le plus mauvais et avec un compilateur médiocre, on obtiendra un temps d'exécution plus rapide que celui qu'on va obtenir si l'on exécute l'algorithme de tri par sélection (ou par insertion), même sur une machine plus puissante, ayant un très bon système d'exploitation et avec un compilateur très performant. Il s'agit ici d'une complexité algorithmique intrinsèque : la fonction $c_1 N^2$ (ou $c_2 N^2$) croit plus vite que la fonction $c_3 N(\log_2 N)$, et peut importe l'ordre de grandeur des constantes.

Il existe plusieurs types d'analyse de la performance d'un algorithme :

- L'analyse optimiste (analyse dans le cas le plus favorable ou analyse dans le meilleur des cas).
- L'analyse en moyenne.
- L'analyse pessimiste (analyse dans le cas le plus défavorable ou analyse dans le pire des cas).

L'analyse dans le meilleur des cas, informe sur le temps minimum de calcul qu'effectue un algorithme. L'analyse en moyenne, permet de connaître le temps moyen de calcul d'un algorithme. Elle renseigne sur le comportement général d'un algorithme. L'analyse dans le pire des cas, informe sur le temps maximum de calcul qu'effectue un algorithme. Ce type d'analyse est le plus courant. L'intérêt de l'analyse dans le pire des cas est d'évaluer les algorithmes dans leur fonctionnement extrême. Aucune mauvaise surprise n'est donc à prévoir, car le pire des cas est connu à l'avance.

L'étude de l'analyse dans le pire et dans le meilleur des cas sont effectuées en calculant les taux asymptotiques de croissance du temps de calcul d'un algorithme en fonction de sa taille d'entrée indépendamment de la machine sur laquelle l'algorithme est exécuté. L'étude de l'analyse en moyenne utilise essentiellement des techniques issues du calcul probabiliste.

2- Étude d'un exemple concret

Considérons le problème suivant : étant donné un entier positif N, décider s'il est premier ou non. Un nombre N est premier s'il est divisible par un nombre autre que 1 et N lui-même. Un algorithme de test de primalité se contentera alors à chercher un diviseur de N. Si un tel diviseur (candidat) existe, N est non premier. Sinon, N est premier. Nous donnons trois algorithmes pour résoudre le problème de test de primalité :

- Le premier algorithme va chercher un candidat depuis 2 jusqu'à $N-1$.
- Le second algorithme cherchera un candidat depuis 2 à $N/2$, puisque tous les diviseurs de N autre que 1 et N se trouvent dans l'intervalle $[2, N/2]$.
- Le troisième algorithme A3 va chercher un candidat entre 2 et $\sqrt{N}$. En effet, si un nombre N est non premier, alors il possède un diviseur $d \leq \sqrt{N}$.

Algorithme (A1) :

```
Fonction    isPrime(N : Entier) : Logique
Variables   d : Entier;
Début
        Pour d = 2 à (N - 1) Faire :
                Si (N est divisible par d) Alors Renvoyer(Faux);
                                           Sinon Renvoyer(Vrai);
                FinSi
        FinPour
Fin
```

Algorithme (A2) :

```
Fonction    isPrime(N : Entier) : Logique
Variables   d : Entier;
Début
        Pour d = 2 à (N / 2) Faire :
                Si (N est divisible par d) Alors Renvoyer(Faux);
                                           Sinon Renvoyer(Vrai);
                FinSi
        FinPour
Fin
```

Algorithme (A3) :

```
Fonction    isPrime(N : Entier) : Logique
Variables   d : Entier;
Début
        Pour d = 2 à √N  Faire :
                Si (N est divisible par d) Alors Renvoyer(Faux);
                                           Sinon Renvoyer(Vrai);
                FinSi
        FinPour
Fin
```

La taille du problème est le nombre de bits pour coder le paramètre N, c'est-à-dire $\lfloor \log_2 N \rfloor + 1$. Cette quantité croit avec la valeur de N. Nous mesurons la complexité algorithmique en fonction de N, au lieu de $\lfloor \log_2 N \rfloor + 1$. L'opération fondamentale pour le problème de test de primalité est le test de la division de N par un nombre inférieur à N. Pour avoir le pire des cas, il faut choisir la configuration « N est premier ». En prenant en considération ces hypothèses, les complexités respectives des trois algorithmes précédents sont :

$$Coût_{A1}(N) = (N-1) - 2 + 1 = N - 2.$$

$$Coût_{A2}(N) = N/2 - 2 + 1 = \frac{N-2}{2}.$$

$$Coût_{A3}(N) = \sqrt{N} - 2 + 1 = \sqrt{N} - 1.$$

La fonction $\sqrt{N}-1$ croit moins vite que la fonction $\frac{N-2}{2}$, qui croit à son tour, moins vite que la fonction $N-2$. Nous pouvons dire que l'algorithme (A3) est plus rapide que (A2), qui à son tour, plus rapide que (A1), lorsqu'il s'agit de les exécuter sur des nombres premiers très grands.

Un exercice de programmation :

Programmer les trois algorithmes précédents (en utilisant un langage de programmation de votre choix). Pour C++, utiliser la bibliothèque standard <time.h> pour mesurer le temps d'exécution des trois algorithmes. Voici le mode d'utilisation :

- Déclarer trois variables t1, t2 et t de type float.
- Écrire l'instruction « t1 = clock( ); » avant le code de l'algorithme.
- Écrire l'instruction « t2 = clock( ); » après le code de l'algorithme.
- Écrire l'instruction « t = (t2 - t1) / CLOCKS_PER_SEC; » pour obtenir le temps d'exécution en secondes.

3- Les grandes familles de complexité

Famille d'algorithmes	Notation	Exemples
Algorithmes **constants**	$\Theta(1)$	Echange deux valeurs
Algorithmes **logarithmiques**	$\Theta(\log n)$	Recherche binaire (dichotomique)
Algorithmes **linéaires**	$\Theta(n)$	Recherche séquentielle
Algorithmes **quasi-linéaires**	$\Theta(n \log n)$	Tri par fusion
Algorithmes **quadratiques**	$\Theta(n^2)$	Tri par sélection
Algorithmes **cubiques**	$\Theta(n^3)$	Produit de deux matrices
Algorithmes **polynomiaux**	$\Theta(n^p),\ p \geq 1$	
Algorithmes **exponentiels**	$\Theta(a^n),\ a > 1$	Calcul récursif de la suite de Fibonacci

4- Les notations asymptotiques

Dans la suite, on ne considère que des fonctions numériques définies de $\mathbb{N}$ dans $\mathbb{R}^+$.

4.1 Notation « grand-O »

Définition. Soit $g(n)$ une fonction positive. On définit l'ensemble $O(g(n))$ par :

$$O(g(n)) = \left\{ f(n) \;\middle|\; (\exists\, c > 0), (\exists\, n_0 \geq 0) \textit{ tels que } : 0 \leq f(n) \leq c.g(n); (\forall n \geq n_0) \right\}.$$

Il s'agit de l'ensemble des fonctions bornées supérieurement par la fonction $g(n)$, à des constances multiplicatives près. Lorsque $f(n) \in O(g(n))$, on dit que la fonction $g(n)$ est une **borne supérieure asymptotique** pour la fonction $f(n)$, et l'on écrit par raison de simplicité : $f(n) = O(g(n))$.

Exemple. Soient $f(n) = n^2 - 180n + 144$ et $g(n) = n^2$. Nous avons $f(n) = \mathrm{O}(g(n))$. En effet, $f(n) = n^2 - 180n + 144 \le n^2 + 180n + 144 \le n^2 + 180n^2 + n^2$ pour tout $n \ge 12$. Donc, $(\exists\, c = 182 > 0)$ et $(\exists\, n_0 = 12 \in \mathbb{N})$ tels que : $f(n) \le cg(n),\ (\forall n \ge n_0)$.

4.2 Notation « grand-Ω »

Définition. Soit $g(n)$ une fonction positive. On définit l'ensemble $\Omega(g(n))$ par :

$$\Omega(g(n)) = \left\{ f(n) \mid (\exists\, c > 0), (\exists\, n_0 \ge 0) \textit{ tels que } : c.g(n) \le f(n); (\forall n \ge n_0) \right\}.$$

Il s'agit de l'ensemble des fonctions bornées inférieurement par la fonction $g(n)$, à des constances multiplicatives près. Lorsque $f(n) \in \Omega(g(n))$, on dit que la fonction $g(n)$ est une **borne inférieure asymptotique** pour la fonction $f(n)$, et l'on écrit par raison de simplicité : $f(n) = \Omega(g(n))$.

Exemple. Soient $f(n) = n^2 - 180n + 144$ et $g(n) = n$. Nous avons $f(n) = \Omega(g(n))$. On a : $f(n) - g(n) = n^2 - 181n + 144$. Montrer que cette différence est positive à l'infini.

Première méthode : $\lim_{n \to \infty +} (n^2 - 181n + 144) = +\infty$. Donc, $n^2 - 181n + 144 \ge 0$, à partir d'un certain rang n_0 (car sinon, on aura nécessairement : $n^2 - 181n + 144 < 0, \forall n \ge 0$. Par suite, $\lim_{n \to \infty +} (n^2 - 181n + 144) \le 0$, ce qui est fausse). Nous avons donc prouvé qu'il existe deux constantes $c = 1 > 0$ et $\exists\, n_0 \in \mathbb{N}$, tels que $f(n) \ge cg(n),\ (\forall n \ge n_0)$.

Deuxième méthode : Considérons le trinôme $n^2 - 181n + 144$. On a : $\sqrt{\Delta} = \sqrt{(-181)^2 - 4 \times 144} \approx 179.40 > 0$. Le trinôme possède alors deux racines : $n_1 \approx \dfrac{181 - 179.40}{2} \approx 0.79$ et $n_2 \approx \dfrac{181 + 179.40}{2} \approx 180.20$. Par suite, $n^2 - 181n + 144 \ge 0$, dès que $n \ge n_2 = 181$. Par conséquent, $(\exists\, c = 1 > 0)$ et $(\exists\, n_0 = 181 \in \mathbb{N})$, pour lesquels on a : $f(n) \ge cg(n),\ (\forall n \ge n_0)$.

4.3 Notation « grand-Θ »

Définition. Soit $g(n)$ une fonction positive. On définit l'ensemble $\Theta(g(n))$ par :

$$\Theta(g(n)) = \left\{ f(n) \mid (\exists\, c_1, c_2 > 0), (\exists\, n_0 \ge 0) \textit{ tels que } : c_1.g(n) \le f(n) \le c_2.g(n); (\forall n \ge n_0) \right\}.$$

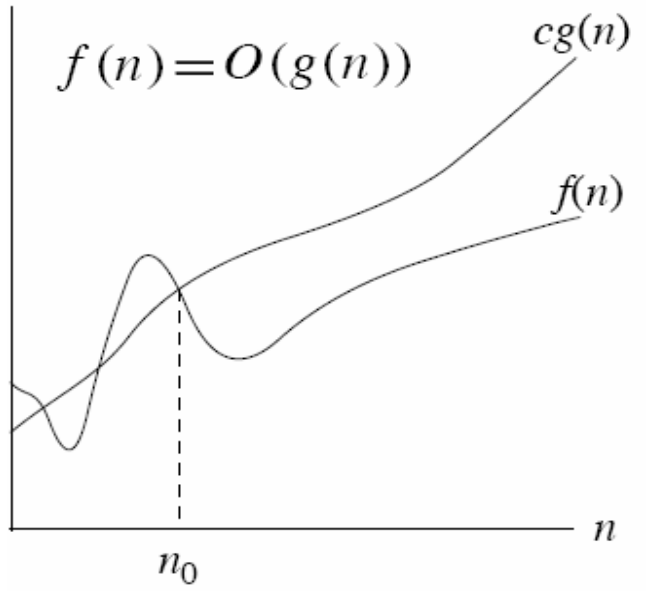
Il s'agit de l'ensemble des fonctions bornées inférieurement et supérieurement par la fonction $g(n)$, à des constances multiplicatives près. Lorsque $f(n) \in \Theta(g(n))$, on dit que la fonction $g(n)$ est une **borne asymptotique** pour la fonction $f(n)$, et l'on écrit par abus de langage, $f(n) = \Theta(g(n))$.

Exemple. Soient $f(n) = n^2 - 180n + 144$ et $g(n) = n^2$. Nous avons $f(n) = \Theta(g(n))$. En effet, on a déjà prouvé que $f(n) \le 182 \times n^2$; pour tout $n \ge 12$. De plus, on a : $f(n) - \dfrac{1}{2} g(n) = \dfrac{1}{2} n^2 - 180n + 144 \ge 0$, à partir d'un certain rang, puisque :

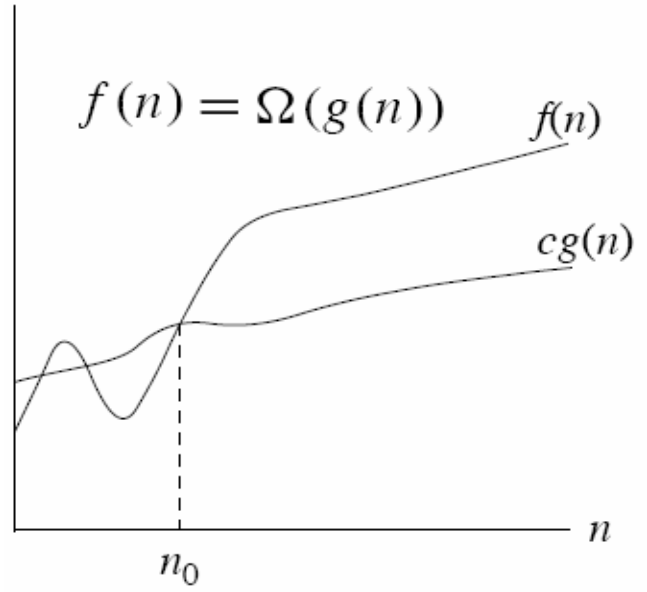
$\lim_{n\to\infty+}(\frac{1}{2}n^2 - 180n + 144) = +\infty$. Finalement, il existe deux constantes positives $c_1 = \frac{1}{2}$ et $c_2 = 182$, et un rang $n_0 \geq 12$ telles que : $c_1 g(n) \leq f(n) \leq c_2 g(n)$; pour tout nombre $n \geq n_0$. Par suite, $f(n) = \Theta(g(n))$.

Remarque. $f(n) = \Theta(g(n)) \Leftrightarrow \big(f(n) = \mathrm{O}(g(n))$ *et* $f(n) = \Omega(g(n))\big)$.

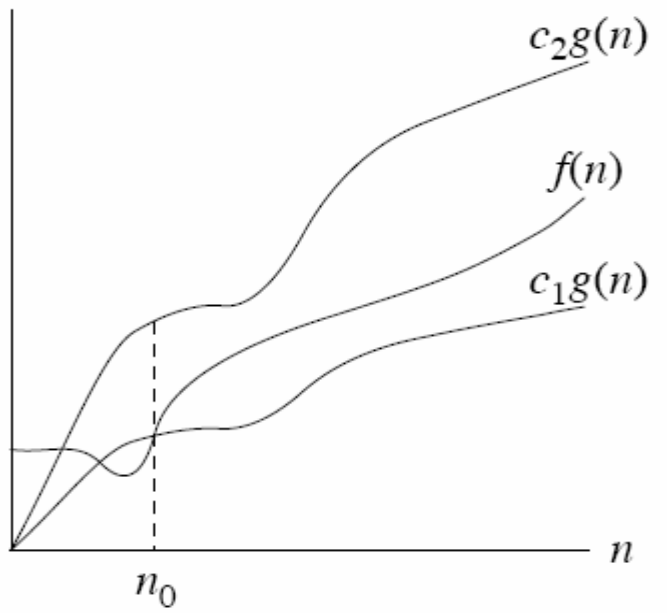
Interprétation géométrique de la notation O :



Interprétation géométrique de la notation Ω :



Interprétation géométrique de la notation Θ :



5- Propriétés des notations asymptotiques

Théorème 1 (Transitivité des notations O, Ω et Θ)

(i) Si $f = \mathrm{O}(g)$ et $g = \mathrm{O}(h)$, alors $f = \mathrm{O}(h)$.

(ii) Si $f = \Omega(g)$ et $g = \Omega(h)$, alors $f = \Omega(h)$.

(ii) Si $f = \Theta(g)$ et $g = \Theta(h)$, alors $f = \Theta(h)$.

Théorème 2 (Réflexivité des notations O, Ω et Θ)

(i) $f = \mathrm{O}(f)$.

(ii) $f = \Omega(f)$.

(ii) $f = \Theta(f)$.

Théorème 3 (Symétrie de la notation Θ)

$f = \Theta(g) \Leftrightarrow g = \Theta(f)$.

Théorème 4 (Symétrie transposée entre la notation Ω et la notation O)

$f = \mathrm{O}(g) \Leftrightarrow g = \Omega(f)$.

6- Notations asymptotiques et fonctions particulières

Une fonction polynomiale de **degré** $d \in IN$ ou tout simplement polynôme, est une fonction définie sur IR par : $f(x) = a_d x^d + a_{d-1} x^{d-1} + ... + a_1 x + a_0$. Les nombres réels a_i pour $i = d, d-1, ..., 1, 0$ sont les **coefficients** du polynôme f. Notons que $a_d \neq 0$ (le

coefficient dominant du polynôme *f*(*x*)). On écrit aussi : $f(x) = \sum_{i=0}^{d} a_i x^i$. Une fonction polynomiale $f(x) = \sum_{i=0}^{d} a_i x^i$ telle que $a_d > 0$ est dite **fonction polynomiale asymptotiquement strictement positif**.

Théorème 5.

Soit $f(n) = \sum_{i=0}^{d} a_i n^i$ un polynôme asymptotiquement strictement positif ($a_d > 0$). Alors : $f(n) = \sum_{i=0}^{d} a_i n^i = \Theta(n^d)$.

Définition. On dit qu'une fonction $f(n)$ admet une **borne asymptotique polynomiale**, si et seulement si $(\exists\, d > 0)$ tel que : $f(n) = \Theta(n^d)$.

Exemple. La fonction *sin*(*n*), *log*(*n*) et les fractions rationnelles admettent des bornes asymptotiques polynomiales.

Théorème 6.

Pour tous nombres réels $a > 1$ et b : $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$.

Ainsi, toute ***fonction exponentielle quelconque*** de base $a > 1$, **croît plus vite** que n'importe quelle ***fonction polynomiale***.

Théorème 7.

Pour tous nombres réels $a > 0$ et $b > 0$: $\lim_{n\to\infty} \frac{(\ln n)^b}{n^a} = 0$.

Ainsi, une ***fonction polynomiale strictement positif*** *quelconque* **croît plus vite** que n'importe quelle ***fonction polylogarithmique***.

Définition. On dit qu'une fonction $f(n)$ admet une **borne asymptotique polylogarithmique**, si et seulement si $(\exists\, a > 0)$ tel que : $f(n) = \mathrm{O}((\ln n)^a)$.

Théorème 8. **(Formule de Stirling pour la factorielle)**

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right).$$

7- Notations asymptotiques et séries numériques usuelles

La série arithmétique :

C'est la série $\sum_{i=1}^{\infty} i = 1 + 2 + ... + n + ...$ Cette série est divergente. La somme partielle de la série arithmétique vérifie la relation suivante : $\boxed{\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2).}$

Les séries géométriques :

Soit x un nombre réel différent de 1. La suite géométrique de raison x est définie par $u_n = x^n$. Sa somme partielle est $\sum_{i=0}^{n} x^i = 1 + x + x^2 + ... + x^n = \frac{x^{n+1} - 1}{x - 1}$.

Théorème 9.

La série géométrique $\sum_{i=0}^{\infty} x^i$ converge si et seulement si $|x| < 1$ et l'on a : $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$.

La série harmonique :

Pour tout entier naturel non nul n, on définit le **n^ème nombre harmonique** par :

$$\boxed{H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}.}$$

Théorème 10.

La série harmonique $\sum_{i=1}^{\infty} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} + ...$ est divergente.

Approximation d'une somme par une intégrale :

Soit $a_i = f(i)$ une suite numérique dont le terme d'indice i peut s'exprimer comme une fonction numérique de la variable i. Soit m et n deux entiers naturels tels que $m < n$. On a le résultat suivant :

Théorème 11.

(i) Si f est croissante, alors $\int_{m-1}^{n} f(x)dx \le \sum_{i=m}^{n} a_i \le \int_{m}^{n+1} f(x)dx.$

(ii) Si f est décroissante, alors $\int_{m}^{n+1} f(x)dx \le \sum_{i=m}^{n} a_i \le \int_{m-1}^{n} f(x)dx.$

La première inégalité se démontre en découpant les intégrales sur l'intervalle [m, n] et en utilisant la relation de Chasles. La deuxième inégalité se déduit de la première, en remarquant que si $f(n)$ est décroissante, alors $-f(n)$ est croissante.

Exemple. Montrons que : $\sum_{i=1}^{n}\frac{1}{i^2}=\mathrm{O}(1)$. Pour cela, il suffit de montrer que $\sum_{i=1}^{n}\frac{1}{i^2}\le c$ pour une constante $c>0$ et à partir d'un certain rang n_0. Posons : $a_i=\frac{1}{i^2}=f(i)$. La fonction $f(x)=\frac{1}{x^2}$ étant décroissante, on peut écrire pour tout entier $n\ge 2$: $\sum_{i=1}^{n}\frac{1}{i^2}=1+\sum_{i=2}^{n}a_i\le 1+\int_1^n\frac{dx}{x^2}=1+\left[\frac{-1}{x}\right]_1^n=2-\frac{1}{n}\le 2$. Par suite, $(\exists\, c=2>0)$, $(\exists\, n_0=2)$ tels que $\sum_{i=1}^{n}\frac{1}{i^2}\le k, (\forall n\ge n_0)$. Ce qui prouve que $\sum_{i=1}^{n}\frac{1}{i^2}=\mathrm{O}(1)$.

8- Résolution des récurrences de partitions

Une équation de récurrence de partitions se présente sous la forme suivante :

$$T(n)=\begin{cases}\Theta(1);\ si\ n=n_0\\ aT\left(\frac{n}{b}\right)+f(n);\ si\ n>n_0\end{cases}$$

Le résultat principal qui permet de résoudre les équations de récurrence de partitions s'appuie sur le théorème général (***Master Theorem***). Ce dernier repose sur le lemme suivant :

Lemme.

Soit c une constante positive et $f(n)=1+c+c^2+c^3+...+c^n=\sum_{i=0}^{i=n}c^i$. Alors :

(i) $f(n)=\Theta(1)$ si $c<1$.

(ii) $f(n)=\Theta(n)$ si $c=1$.

(iii) $f(n)=\Theta(c^n)$ si $c>1$.

Théorème 12 (Le théorème général).

Si $T(n)=\begin{cases}\mathrm{O}(1);\ si\ n=1\\ aT\left(\frac{n}{b}\right)+\mathrm{O}(n^d);\ si\ n>1\end{cases}$ pour certaines constantes $a>0, b>1$ et $d\ge 0$,

alors : $T(n)=\begin{cases}\mathrm{O}\left(n^d\right);\ si\ d>\log_b a\\ \mathrm{O}\left(n^d\log_b n\right);\ si\ d=\log_b a\\ \mathrm{O}\left(n^{\log_b a}\right);\ si\ d<\log_b a\end{cases}$

Mots clés du chapitre :

Algorithme, Complexité d'un algorithme, Notations asymptotiques (O, Ω et Θ), Approximation par intégrales, Méthode de découpage d'une sommation, Récurrence de partitions, Le théorème général.