USTHB - 2010/2011 **Master I**
DÉPT. INFORMATIQUE ***N.Bensaou***

Complexité et algorithmique avancé

Table des matières

Chapitre 1

Introduction

En informatique, deux questions fondamentales se posent toujours devant un problème à résoudre :

- existe t-il un algorithme pour résoudre le problème en question ?
- si oui, cet algorithme est-il utilisable en pratique, autrement dit le temps et l'espace mémoire qu'il exige pour son exécution sont-ils "raisonnables" ?

La première question a trait à la **calculabilité**, la seconde à la **complexité**.

1.1 La recherche en logique et en mathématique : un bref historique

En calculabilité on cherche à définir les problèmes dont les solutions sont calculables par un algorithme et ceux qui ne le sont pas. Ces derniers sont les problèmes pour lesquels il n'existe pas de solution (algorithme) quel que soit les progrès futurs de la technologie.
La théorie de la calculabilité est née à la fin du 19-ıème siècle de questions de logique et de mathématique. On cherchait à connaître les limites de la connaissance et la prouvabilité d'une vérité mathématique donnée.

Résoudre un problème en logique revient à donner une démonstration d'une proposition quelconque. Dans la cadre du calcul propositionnel, on disposait depuis 1934 de méthodes de preuves de validité (les systèmes de déduction naturelle de Gentzen) ce qui établit la décidabilité du calcul propositionnel. Dans le cadre du calcul des prédicats, il y a des quantificateurs et l'application des règles de déduction peut se continuer à l'infini.
Au début du 20ème siècle, David Hilbert avait pour projet de rechercher les principes de base des mathématiques. Il se propose de :

1. formaliser toutes les mathématiques ;

2. trouver les axiomes et les règles de déductions qui permettraient de démontrer toutes les vérités mathématiques ;
3. mécaniser (automatiser) le raisonnement mathématiques.

Mais en 1931, le mathématicien Kurt Gödel publie son théorème d'incomplétude qui démontre les limites de ce projet : "dans n'importe quel système formel finiment axiomatisé, il existe une proposition indécidable", autrement dit, une proposition pour laquelle on ne saura pas démontrer ni sa vérité ni sa fausseté.

Le projet de Hilbert s'avère donc être un projet impossible. Ainsi :

1. tout n'est pas démontrable ; et
2. on ne peut pas donner une liste finie de tous les principes permettant de développer une preuve mathématique.

1.2 Conséquences du théorème d'incomplétude sur l'informatique

• D'un point de vue philosophique les conséquences du théorème d'incomplétude de Gödel signifient que :

1. La vérité est plus large que la démontrabilité ;
2. on ne peut pas construire toutes les vérités de façon mécanique.

La preuve du théorème de Gödel est basée sur le *paradoxe du menteur* qui permet de construire une proposition qui dit d'elle-même qu'elle est indémontrable. Plus formellement, elle s'exprime ainsi :

Soit S un système formel cohérent et soit A la formule suivante : " A est indémontrable dans S". Cette formule est telle que ni A, ni sa négation non-A ne pourront être formellement démontrées dans S.

• D'un point de vue informatique, le théorème d'incomplétude de Gödel signifie une impossibilité fondamentale. Tout problème ne peut être résolu par un algorithme. Il est donc impossible de concevoir un programme capable de vérifier tous les programmes. Pour prouver ce résultat, on suppose qu'un programme P capable de vérifier tous les programmes existe et on définit un programme B et on montre que P ne peut pas le vérifier.

Preuve :

Supposons qu'un programme capable de vérifier tous les programmes existe. Soit P ce programme.

1) Pour tout programme A, s'il est juste alors on écrit que P(A) = v (« vrai ») et s'il est faux on écrit P(A) = f (« faux »).

2) Soit B le programme suivant : $\mathbf{B = \ll P(B) = f \gg}$.

3) Si P(B) = v (le programme B est juste), alors le programme P(B) = f est faux ; donc P[P(B) = f] = P(B) = f, ce qui est contraire à l'hypothèse. Si P(B) = f, alors on a P[P(B) = f] = P(B) = v, ce qui est encore contraire à l'hypothèse.

4) Ainsi B ne peut pas être vérifié par P. Il ne peut donc pas exister de programme capable de vérifier tous les programmes.

Avec l'apparition des ordinateurs, en décidabilité, on s'intéresse à étudier les limites de l'informatique indépendamment des machines qui les exécutent. Il s'agit de trouver quels sont les problèmes qui sont insolubles (ou indécidables) et qui le seront toujours quelque soit le développement technologique futur. Pour les problèmes décidables, une seconde nécessité apparaît : il n'était plus suffisant de savoir qu'une solution existe (la décidabilité du problème), mais il devenait impératif de savoir que la solution soit réalisable, exécutable" de manière "raisonnable". C'est le domaine de la complexité des algorithmes ou encore l'analyse des algorithmes.

Ce théorème implique ainsi une dichotomie entre les problèmes qui ont une solution algorithmique et ceux qui n'en ont pas. La recherche s'est développée par la suite dans ces deux directions. Pour les problèmes qui ont une solution algorithmique, on cherchera à les comparer et les classer en une hiérarchie en fonction de leur complexité, une fonction qui mesure les ressources (temps ou espace) nécessaires à l'exécution de l'algorithme en question.

1.3 Exemple

Certes on peut définir des méthodes de vérification efficaces et les outiller de sorte qu'elles soient faciles à utiliser. Mais ces méthodes, aussi efficaces soient-elles, ne garantissent pas que tous les programmes qu'elles valident soient corrects. La ressource temps mesure le nombre d'opérations élémentaires nécessaires en fonction de la taille des données. La taille des données s'exprime en nombre de caractères nécessaire pour décrire une donnée du problème.

Généralement le temps de calcul d'un algorithme croît avec la taille des données. La complexité est la vitesse de cette croissance.

Une croissance exponentielle rend un problème intraitable pour les données de grande taille. Une croissance polynomiale, avec une puissance

au dessus de la puissance 3 ou 4 rend aussi le problème intraitable pour des données de grande taille.

Exemple 1.3.1 *Générer toutes les permutations de n éléments.*

La complexité est plus que exponentielle puisqu'il y a n! permutations à faire. Si n est grand le temps est énorme !

Chapitre 2

Les bases de l'analyse d'un algorithme

2.1 Problème, instance, solution et programme

Le but en analyse d'algorithmes est d'étudier le fonctionnement d'un algorithme. Il s'agit, pour les problèmes solubles, d'estimer les ressources nécessaires au déroulement de la solution considérée. Les principales ressources qui nous intéressent sont les fonction temps d'exécution et espace mémoire nécessaire. Pour formaliser ces notions on doit distinguer la notion de problème de celle de programme.

2.1.1 Notion de problème

Un problème est une question qui a les propriétés suivantes :

1. elle est générique et s'applique à un ensemble d'éléments ;
2. toute question posée pour chaque élément admet une réponse ;

Exemple 2.1.1 *1. Déterminer si un entier donné est pair ou impair ;*

2. déterminer le maximum d'un ensemble d'entiers donnés

3. trier une suite d'entiers donnés, etc.

2.1.2 Notion d'instance

Une instance de problème est la question générique appliquée à un élément :

Exemple 2.1.2 – *l'entier 15 est-il pair ou impair ?*
- *déterminer le maximum de* $\{1, 5, -14, 0, -5\}$.
- *trier la suite précédente.*

En informatique, le concept de *problème* est formalisé par celui de *langage* : les instances d'un problème peuvent être représentées par des mots (chaînes de caractères). La solution d'un problème donné est un objet qu'il faut determiner et qui satisfait les contraintes explicite imposées dans le problème. Il existe quatre manières d'obtenir la solution d'un problème donné :

1. appliquer une formule explicite qui donne la solution ; (rechercher les solutions de l'équation du second degré)
2. utiliser une définition récursive ;
3. utiliser un algorithme qui converge vers la solution ;
4. énumérer des cas (par essai et erreur).

2.1.3 Notion de problème binaire

Un problème binaire (oui/non) est un ensemble d'éléments divisés en deux parties : les instances positives et les instances négatives. Les instances du problème "déterminer si un entier n est pair ou impair" sont les entiers (qui existent indépendamment de leur représentation). L'ensemble, caractérisé par ses opérations, peut être partitionné en deux sous ensembles : pair et impair.

On considère les problèmes dont les instances sont encodées par des mots définis sur un alphabet. L'ensemble des mots est partitionné en deux :
- les mots représentants les instances positives (réponse oui)
- les mots représentants les instances négatives (réponse non) ou ne représentant pas d'instance.

Un problème est ainsi caractérisé par les encodages de ses instances positives. Résoudre un problème c'est reconnaître les instances positives de ses encodages.

Exemple 2.1.3 *1. l'ensemble de toutes les représentations binaires de nombres paires est un langage*

2. l'ensemble des mots représentant des programmes écrits en pascal qui s'arrêtent toujours est un langage.

Pour savoir décrire les problèmes il suffit de savoir décrire les langages. Si le langage est fini, il suffit d'énumérer ses mots pour le décrire. Si le langage est infini, il faut appliquer les opérations sur les langages [1].

1. (cf. théorie des langages)

2.1.4 Solution

Définition 2.1.4 *Une procédure effective est définie en termes de langage* décidé *(accepté) par une machine de Turing. Pour démontrer que certains problèmes ne sont pas solubles par une procédure effective, on prouve que les langages qu'ils encodent ne sont pas décidés par une machine de Turing.*

Une solution à un problème pouvant être exécuté par un ordinateur est une *procédure effective.*

Si un problème est récursif alors il existe une procédure qui le résoud. Mais rien ne garantit que cette procédure soit utilisable pratiquement : le temps et l'espace mémoire nécessaire à son exécution pourraient largement dépasser ce dont on peut espérer disposer. Un algorithme utilisable se doit d'être ***efficace***.

2.1.5 Notion de programme

Cette notion va permettre de distinguer les solutions qui pourront être exécutées par un ordinateur ou "procédure effective" des solutions non effectives (qui ne s'arrêtent pas).

Un programme c'est donc la solution d'un problème pouvant être exécutée par un ordinateur. Il contient toute l'information nécessaire pour résoudre le problème **en temps fini**. L'exemple suivant donne une idée d'une procédure non effective.

Exemple 2.1.5 *Déterminer si un programme donné n'a pas de boucles ou de séquences d'appels récursifs infinies.*

Il n'y a aucun moyen pour savoir si une boucle est infinie ou non.

2.2 L'analyse d'un algorithme

L'analyse des algorithmes s'exprime en termes d'évaluation de la complexité de ces algorithmes. On présente les éléments de base de la complexité.

Définition 2.2.1 *Un algorithme est un ensemble fini d'instructions qui, lors de leur exécution, accomplissent une tâche donnée.*

Définition 2.2.2 *[?] Un algorithme est un ensemble d'opérations de calcul élémentaires, organisé selon des règles précises dans le but de résoudre un problème donné.*

Un algorithme doit satisfaire les propriétés suivantes :

Remarque 2.2.3 *Propriétés*

1. *il peut avoir zéro ou plusieurs quantités de données en entrée ;*
2. *il doit produire au moins une quantité en sortie ;*
3. *chacune de ses instruction doit être claire et non ambiguë ;*
4. *il doit terminer après un nombre finie d'étapes ;*
5. *chaque instruction doit être implémentable ;*

Exemple 2.2.4 *Soit le problème : "trier un ensemble de $n \geq 1$ d'entiers".*

Une solution pourrait être la suivante :
"*Parmi les entiers qui ne sont pas encore triés, rechercher le plus petit et le placer comme successeur dans la liste triée. La liste triée est initialement vide*".
Cette solution n'est pas un algorithme car elle laisse plusieurs questions sans réponse. Elle ne dit pas où ni comment les entiers sont initialement triés ni où doit être placé le résultat.

On supposera que les entiers sont dans un tableau t tel que le i-ème entier soit stocké à la ième position : $t[i], 0 \leq i < n$. L'algorithme suivant est la solution d'un tri par sélection.

```
for (i=0; i<n; i++) {
calculer le le min de t, de i à n-1, et supposer que le plus petit est en t[min]
permuter t[i]  et t[min];
}
```

En algorithmique (étude des algorithmes), en dehors de l'analyse, on peut s'intéresser aux questions suivantes :

- la modularité ;
- la correction ;
- la maintenance ;
- la simplicité ;
- la robustesse ;
- l'extensibilité,
- le temps de mise en oeuvre,
- etc.

2.2.1 Qu'est-ce que l'analyse

L'analyse d'un algorithme consiste à déterminer la quantité de ressources nécessaires à son exécution. Ces ressources peuvent être la quantité de mémoire utilisée, la largeur d'une bande passante, le temps de calcul etc. Nous nous intéressons dans ce cours au temps et à la mémoire. Le but est de pouvoir identifier, face à plusieurs algorithmes qui résolvent un même problème, celui qui est le plus efficace.

L'étude des performances des algorithmes a plusieurs avantages :
- l'algorithmique nous aide à comprendre la scalabilité (portabilité sur des plates-formes de taille plus réduite ou plus grande)
- les performances limitent une frontière entre ce qui est faisable et ce qui est impossible à faire ;
- Les mathématiques de l'algorithmique fournissent un langage pour parler du comportement des programmes ;
- les conséquences tirées des performances des programmes peuvent se généraliser à d'autres ressources informatiques ;
- les performances sont le crédit de l'informatique ;
- la rapidité est une performance recherchée dans la plupart de nos activités, elle a un caractère agréable !

2.3 Paramètres de l'analyse

Les performances d'un algorithme dépendent de trois principaux paramètres suivants :
- **La taille des entrées**
  Il est intuitivement évident, par exemple, que les séquences courtes sont plus faciles à trier que les séquences longues La complexité est donc paramétrée par la taille des donnés puisque . Ce paramètre taille dépend du problème étudié, mais très souvent c'est le nombre d'éléments constituant l'entrée. C'est par exemple la longueur d'une liste pour un algorithme de recherche ou de tri, le nombre total de bits nécessaire pour représenter l'entrée pour un produit de matrices. Ce paramètre peut être composé de plus d'une valeur. Pour un graphe, l'entrée est le nombre de noeuds et le nombre d'arcs. Pour chaque problème il faut en premier lieu caractériser l'entrée.
- **La distribution des données**
  Un tableau déjà trié est facile à trier et un tableau trié en sens inverse de l'ordre cherché est le plus long à réaliser. Mais cette information est généralement difficile à obtenir.

– **La structure de données**
  La représentation des données un paramètre fondamental. Une donnée directement accessible (l'élément d'un tableau) est plus rapidement obtenu qu'un élément dans une structure à accès séquentiel comme les listes.

2.3.1 Types de complexité

Il existe trois types de complexité :
– complexité du meilleur cas
  C'est le cas, pour l'exemple du tri, où la suite initiale est triée. Cette mesure n'a pas d'intérêt puisque tous les algorithmes réagissent de la même manière. Elle ne permet pas de distinguer deux solutions.
– complexité en moyenne
  Elle nécessite la connaissance de la distribution des données autrement dit des fonctions de probabilités sur les données, qui peuvent être obtenues après avoir effectué plusieurs tests.
– complexité du cas pire
  Elle fournit une borne supérieure du temps d'exécution qui indique que dans toutes les situations, l'algorithme ne peut pas dépasser la valeur de cette borne. Elle constitue donc une **garantie**, ce que nous cherchons toujours ! C'est donc à cette complexité que nous nous intéressons dans la suite de ce cours.

Il s'agit donc de trouver une équation qui relie le temps d'exécution à la taille des données. Nous pourrons ainsi comparer deux algorithmes qui résolvent le même problème en comparant le rapport de leur équation et choisir le plus rapide des deux.

2.4 La complexité asymptotique

En pratique, il est difficile de calculer de manière exacte la complexité d'un algorithme. Si n est la donnée d'entrée, on se limite à une évaluation *asymptotique* de l'algorithme. La complexité asymptotique est une *approximation* du nombre d'opérations que l'algorithme exécute en fonction de la *donnée entrée* : asymptotique car elle prend en compte une donnée de grande taille et ne retient que le terme de poids fort dans la formule et ignore le coefficient multiplicateur.

Exemple 2.4.1 *Soit n la donnée de grande taille et $T(n)$ la complexité telle que $T(n) = 10 * n^3 + 3 * n^2 + 5 * n + 1$, la complexité asymptotique est : $T(n) = O(n^3)$.*

2.5 Les notations de Landau

- La notation O
- La notation Θ
- La notation o

Exemple 2.5.1 *calculer la somme 1+2+3+...+N*

2.6 Types d'algorithmes

Un algorithme est soit itératif soit récursif. L'algorithme itératif est constitué de structures de contrôles : les boucles et le choix. L'analyse consiste à estimer le nombre de répétitions des itérations. Les algorithmes récursifs consistent à ramener la solution d'un problème de taille n à celle d'un nombre réduit de sous problèmes de même type que le problème initial mais de taille $m < n$. L'analyse d'un algorithme récursif consiste à estimer le temps d'exécution de tous les sous problèmes et leur composition pour résoudre le problème en entier. La combinaisons des analyses de chaque sous problème fournit une r*elation de récurrence* qui sera transformée en une équation qui relie le nombre d'opérations total à la taille du problème initial.

Exemple 2.6.1 *Calcul du maximum de quatre valeurs a, b, dc, d :*
Solution1 :

```
int maximum (int a, int b, int c, int d) {

int max = a;

if (b > max) max = b;
if (c > max) max = c;
if (d > max) max = cd;

return max;
}
```

Solution2 :

```
int maximum (int a, int b, int c, int d) {

if (a > b)
    if(a > c)
```

```
        if (a > d) return a;
                else return d;
          else
             if (c > d) return c;
                else return d;
        if (b >c )
            if (b > d) return b;
                     else return d;
         else
           if (c > d) return c;
                else return d;
}
```

Les deux solutions exigent exactement 3 comparaisons pour déterminer la réponse bien que la solution 1 soit plus simple à comprendre. Elles sont de même complexité temporelle pour une exécution sur machine. Mais en termes d'espace, la solution1 exige un espace supplémentaire pour stocker le max. La quantité d'espace supplémentaire étant insignifiante, ces deux solutions sont aussi équivalentes en complexité spatiale. L'évaluation de l'efficacité d'un algorithme en termes d'espace occupé et la comparaison entre algorithmes se fait toujours pour des grandes tailles de données.

L'analyse de performances d'un algorithme est indépendante du type de la machine sur laquelle il s'exécute. Elle ne fournit pas le nombre exact d'opérations mais un ordre de grandeur. En analyse d'algorithme, il n'y a pas de différences entre une solution qui fait N opérations et une autre qui fait $N+300$ opérations, car N et $N+300$ tendent vers la même limite quand N tend vers l'infini.

Exemple 2.6.2 *Calcul du maximum d'une suite.*

```
int maximum (tab t, int n) {

int max = t[0];

for (i=1; i<n; i++)
if (t[i] > max) max = t[i];


return max;
}
```

L'analyse :

1. Si la liste est triée en ordre décroissant on ne fera qu'une seule affectation, celle qui est avant la boucle.
2. Si la liste est triée en ordre décroissant, on fera n affectations, une avant la boucle et $n-1$ dans la boucle.
3. Si $n = 10$, il y a 10! façons d'arranger ces nombres. Si le max est en première position, on fera une affectation, s'il est en 2eme position, on en fera deux, s'il est en 3eme position, on en fera 3 et ainsi de suite. S'il est en dernière position on fera n affectations.

2.7 La complexité temporelle

2.7.1 Le temps d'exécution

Pour analyser un algorithme on se donne un modèle d'analyse. Généralement ce modèle est ***la machine à accès aléatoire (RAM)***. Dans ce modèle les instructions sont exécutés séquentiellement sans opérations simultanées. L'analyse fait appel à des outils mathématiques, comme l'algèbre, l'algèbre combinatoire discrète, la théorie élémentaire des probabilités etc. Sachant qu'un algorithme peut se comporter différemment pour chaque valeur d'entrée possible, il es nécessaire de limiter ce comportement à quelques formules simples et faciles à comprendre.

C'est le nombre d'opérations élémentaires sur une entrée particulière. On peut résumer les règles à respecter pour l'évaluation de la complexité temporelle d'un algorithme comme suit :

1. la complexité des affectations, opérations de lecture/écriture ou comparaison peut se mesurer en O(1) ;
2. la complexité d'une séquence d'instruction est déterminée par la règle de la somme. Elle est égale, à une constante multiplicative près, à la complexité la plus forte parmi toutes les instructions de la séquence ;
3. la complexité d'une instruction

Chapitre 3

Outils mathématiques

L'étude de la complexité algorithmique fait appel à des outils mathématiques (théorie des probabilités, algèbre, algèbre combinatoire discrète, etc.). Calculer la complexité d'un algorithme consiste à determiner une *formule mathématique* qui exprime le coût de cet algorithme. Cette formule s'obtient en bornant des expressions de sommations et/ou de produits par un calcul de limite à l'infini de fonctions. Dans plusieurs situations les propriétés relationnelles entre les nombres réels s'appliquent aussi aux fonctions. Ce chapitre donne quelques rappels fondamentaux liés à l'utilisation des fonctions.

3.1 Sommation

Lorsqu'un algorithme contient une structure de contrôle répétitive, le temps d'exécution peut s'exprimer comme une somme des temps pour exécuter chaque instruction dans le corps de la boucle. Déterminer une fonction de complexité asymptotique exige le calcul d'expressions de sommation de série et savoir les borner.

3.1.1 Formules et propriétés des sommations

Soit une séquence de nombres $a_1, a_2, a_3, \ldots$, la somme infinie $a_1 + a_2 + \cdots + a_n + \ldots$

$$\sum_{k=1}^{\infty} a_k \tag{3.1}$$

est interprétée par la limite suivante :

$$\lim_{n \longrightarrow \infty} \sum_{k=1}^{k=n} a_k \tag{3.2}$$

- Si la limite n'existe pas, la série diverge, autrement elle converge.
- Si n n'est pas entier on suppose que la limite supérieure est $\lfloor n \rfloor$. [1]
- Si la somme commence avec $k = x$ et x n'est pas entier on supposera que la sommation commence avec l'entier $\lfloor x \rfloor$.

Linéarité

Soient $a_1, a_2, \ldots, a_n$ et $b_1, b_2, \ldots, b_n$ deux séquences de nombres et c un réel donné. La linéarité est une propriété définie par l'expression suivante : :

$$\sum_{k=1}^{k=n} (c * a_k + b_k) = c * \sum_{k=1}^{k=n} a_k + \sum_{k=1}^{k=n} b_k. \tag{3.3}$$

Cette propriété est aussi vérifiée pour les séries infinies convergentes. Elle est utile dans la manipulation des sommations qui contiennent des notations asymptotiques.

Exemple 3.1.1

$$\sum_{k=1}^{k=n} (\Theta(f(k)) = \Theta(\sum_{k=1}^{k=n} f(k)) \tag{3.4}$$

La fonction Θ s'applique à la variable k dans le membre gauche et à n dans le membre droit. Cette manipulation s'applique aussi aux séries infinies convergentes.

Séries arithmétiques

$$\sum_{k=1}^{k=n} k = 1 + 2 + 3 + \cdots + n = \frac{n * (n+1)}{2} = \Theta(n^2) \tag{3.5}$$

1. l'expression $\lfloor a \rfloor$ désigne le plus grand entier inférieur ou égal à a et $\lceil a \rceil$ désigne le plus petit entier supérieur ou égal à a.

Séries géométriques

Si x est un réel, $x \neq 1$, on a :

$$\sum_{k=0}^{k=n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{(x^{n+1} - 1)}{x - 1} \tag{3.6}$$

est une série exponentielle (ou géométrique). Si $|x| < 1$ et la sommation est infinie, on obtient la série géométrique décroissante suivante :

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \tag{3.7}$$

3.1.2 Séries harmoniques

Pour des entiers n positifs, le $n^{\text{ième}}$ nombre harmonique vaut :

$$\begin{aligned} H_n =& 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\ =& \sum_{k=1}^{k=n} \frac{1}{k} \\ =& \ln n + O(1). \end{aligned} \tag{3.8}$$

3.1.3 Séries emboîtées

Pour toute série quelconque $a_0, a_1, a_2, \ldots, a_n$ on écrit :

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0. \tag{3.9}$$

car chaque terme de $a_1, a_2, \ldots, a_{n-1}$ est ajouté et supprimé une seule fois. De même que

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n. \tag{3.10}$$

Exemple 3.1.2

$$\sum_{k=1}^{n-1} \frac{1}{k * (k+1)} \tag{3.11}$$

On réécrit chaque terme sous la forme

$$\frac{1}{k*(k+1)} = \frac{1}{k} - \frac{1}{k+1} \tag{3.12}$$

d'où :

$$\begin{aligned} \sum_{k=1}^{n-1} \frac{1}{k*(k+1)} &= \sum_{k=1}^{n-1} (\frac{1}{k} - \frac{1}{k+1}) \\ &= a_0 - a_n \\ &= 1 - \frac{1}{n} \end{aligned} \tag{3.13}$$

3.1.4 Les produits

Le produit fini $a_1 * a_2 * a_3 * \cdots * a_n$ s'écrit :

$$\prod_{k=1}^{k=n} a_k \tag{3.14}$$

- Si $n = 0$ le produit vaut 1 par définition.
- Une formule de produit peut être convertie en une formule de sommation en passant par le logarithme :

$$\log(\prod_{k=1}^{k=n} a_k) = \sum_{k=1}^{k=n} \log a_k. \tag{3.15}$$

Exemple 3.1.3 *Exercices*

- *Trouver une formule simple pour :*

$$\sum_{k=1}^{k=n} (2 * k - 1). \tag{3.16}$$

- *En utilisant les séries harmoniques, montrer que*

$$\sum_{k=1}^{k=n} \frac{1}{2*k-1} = \ln(\sqrt{n}) + O(1) \tag{3.17}$$

- *En utilisant la série 3.7 et la propriété de convergence des dérivées de série convergente, calculez la somme suivante :*

$$\sum_{k=0}^{k=n} k * x^k. \tag{3.18}$$

3.2 Récurrences

3.2.1 Borner une sommation

La récurrence est une des techniques que l'on peut utiliser pour borner une sommation. Par exemple, on peut démontrer par récurrence que $\sum_{k=1}^{k=n} k = \frac{n*(n+1)}{2}$.

La récurrence peut aussi être utiliser pour prouver l'existence d'une borne.

Exemple 3.2.1 *Prouver que* $\sum_{k=0}^{k=n} 3^k$ *est en* $O(3^n)$.

On montre, par récurrence, qu'il existe une constante c *et un entier* n_0 *telles que :*

$$\sum_{k=0}^{k=n} 3^k \leq c * 3^n, \forall n \geq n_0. \tag{3.19}$$

- ***cas de base*** *: pour n=0 on a :* $\sum_{k=0}^{k=n} 3^k = 3^0 = 1 \leq c * 1$ *pour* $c \geq 1$
- ***hypothèse de récurrence*** *: on suppose la formule 3.19 est vraie à l'ordre* n *et montrons qu'elle reste vraie à l'ordre* $n+1$.
- $$\sum_{k=0}^{k=n+1} 3^k = \sum_{k=0}^{k=n} 3^k + 3^{n+1} \leq c * 3^n + 3^{n+1}$$

  Or $c * 3^n + 3^{n+1} = c * 3^{n+1}(\frac{1}{3} + \frac{1}{c}) \leq c * 3^{n+1}$ *si* $\frac{1}{3} + \frac{1}{c} \leq 1$ *ce qui est vrai si* $c \geq \frac{3}{2}$. *C.Q.F.D*

Remarque 3.2.2 *Attention aux erreurs !*

Dans la définition de O *les constantes* c *et* n_0 *ne doivent pas varier avec* n.

Contre-exemple : l'affirmation suivante : $\sum_{k=1}^{k=n} k = O(n)$ *est fausse. La démonstration par récurrence suivante est fausse :*

- ***cas de base*** *: pour n=0 on a :* $\sum_{k=1}^{k=1} k = 1 \leq c * 1$ *vraie pour* $c \geq 1$
- *hypothèse de récurrence :* $\exists c, n_0$ *telles que* $\sum_{k=1}^{k=n} k \leq c * n, \forall n \geq n_0$
- *Calculons* $\sum_{k=1}^{k=n+1} k = \sum_{k=1}^{k=n} k + (n+1) \leq c*n+(n+1) = n*(c+1)+1$. *La constante* c *croit avec* n.
- *Calculons* $\sum_{k=1}^{k=n+2} k = \sum_{k=1}^{k=n} k + (n+1) + (n+2) \leq c * n + (n+1) + (n+2) = n*(c+2)+3$, *on voit bien que la constante* c *n'est pas unique pour tout* $n > n_0$.

3.2.2 Borner un terme par le plus grand terme de la série

On peut borner une sommation en bornant chacun de ses termes par une borne supérieure intéressante. Il suffit dans plusieurs cas de borner tous les termes par le terme le plus grand, autrement dit :

$$\sum_{k=1}^{k=n} a_k \leq n * max\{a_1, a_2, \ldots, a_n\} \tag{3.20}$$

Exemple 3.2.3

$$n = 1 * 2 * 3 * \cdots * n \leq n * n * n * \ldots n = n^n. \tag{3.21}$$

$$\sum_{k=1}^{k=n} k \leq \sum_{k=1}^{k=1} n = n^2. \tag{3.22}$$

3.2.3 Borner une série par une série géométrique

Pour borner une série $\sum_{k=0}^{k=n} a_k$ par une série géométrique, on doit montrer qu'il existe une **constante** r telle que $r < 1$ et $a_{k+1}/a_k \leq r$. Si un tel r existe, alors $a_k \leq a_0 * r^k, \forall k \geq 0$. On obtient ainsi :

Exemple 3.2.4

$$\begin{aligned} \sum_{k=0}^{k=n} a_k \leq & \sum_{k=0}^{\infty} a_0 * r^k = \\ & a_0 * \sum_{k=0}^{\infty} r^k = \\ & a_0 * \frac{1}{1-r}. \end{aligned} \tag{3.23}$$

Cette technique permet d'obtenir une meilleure borne que celle qui utilise le plus grand terme.

Exemple 3.2.5 *Trouver une borne pour* $\sum_{k=1}^{\infty} k/3^k$

- *Pour* $k = 1$, $a_1 = 1/3$

– *le rapport entre deux termes quelconques vaut :*

$$\begin{aligned} a_{k+1}/a_k = & \frac{(k+1)/3^{k+1}}{k/3^k} \\ & \frac{1}{3} * \frac{k+1}{k} \\ & \frac{1}{3} * (1 + \frac{1}{k}) \leq \frac{2}{3} \qquad \forall k \geq 1. \end{aligned} \tag{3.24}$$

Tout terme a_k est donc borné par $a_0 * r^k = (1/3) * (2/3)^k$, d'où :

$$\begin{aligned} \sum_{k=1}^{\infty} k/3^k \leq & \sum_{k=1}^{\infty} (1/3) * (2/3)^k \\ & 1/3 * \frac{1}{1 - 2/3} = 1. \end{aligned} \tag{3.25}$$

Remarque 3.2.6 *Pour appliquer cette méthode il est nécessaire de trouver $r < 1$ constant. Si r varie la méthode n'est pas applicable, comme pour la série divergente suivante :*

$$\sum_{k=1}^{\infty} 1/k = \lim_{n \longleftarrow \infty} \sum_{k=1}^{n} 1/k = \lim_{n \longleftarrow \infty} \Theta(\ln n) = \infty$$

Dans ce cas le rapport $a_{k+1}/a_k < r$ mais un r variable qui tend vers 1. Il n'y a pas de r constant, $r < 1$.

On peut appliquer cette méthode à partir d'un certain rang, à partir duquel il existe un r constant qui vérifie les conditions d'existence d'une série géométrique.

Exemple 3.2.7 *Par exemple, pour borner la série $\sum_{k=1}^{\infty} \frac{k^2}{2^k}$, on calcule la rapport a_{k+1}/a_k. On peut montrer que $a_{k+1}/a_k \leq 8/9$ si $k \geq 3$.*

$$\begin{aligned} \sum_{k=1}^{\infty} \frac{k^2}{2^k} = & \sum_{k=1}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \\ & \leq O(1) + 9/8 \sum_{k=3}^{\infty} (8/9)^k \\ & O(1). \end{aligned} \tag{3.26}$$

3.3 Calcul de sommes par intégrales

Définition 3.3.1 – *Une fonction est monotone croissante si* $m \leq n \Longrightarrow f(m) \leq f(n)$.
– *Une fonction est monotone décroissante si* $m \leq n \Longrightarrow f(m) \geq f(n)$.

Soit une expression de sommation $\sum_{k=m}^{n} f(k)$ à borner.

– Si f est une fonction **monotone et croissante** on a :

$$\int_{m-1}^{n} f(x)dx \leq \sum_{m}^{n} f(k) \leq \int_{m}^{n+1} f(x)dx \tag{3.27}$$

– Si f est une fonction **monotone et décroissante**, on a :

$$\int_{m}^{n+1} f(x)dx \leq \sum_{m}^{n} f(k) \leq \int_{m-1}^{n} f(x)dx \tag{3.28}$$

Exemple 3.3.2 *Bornons* $\sum_{k=1}^{n} \frac{1}{k}$ *par cette méthode. Puisque* $f(k) = 1/k$ *est monotone décroissante, alors :*

$$\int_{1}^{n+1} \frac{1}{x}dx \leq \sum_{1}^{n} \frac{1}{k} \leq \int_{0}^{n} \frac{1}{x}dx \tag{3.29}$$

Nous avons alors d'une part :

$$\begin{aligned} \sum_{k=1}^{n} \frac{1}{k} &\geq \int_{1}^{n+1} \frac{dx}{x} \\ &\geq \ln(n+1). \end{aligned} \tag{3.30}$$

et d'autre part :

$$\begin{aligned} \sum_{k=1}^{n} \frac{1}{k} &= 1 + \sum_{k=2}^{n} \frac{1}{k} \\ &\leq 1 + \int_{1}^{n} \frac{dx}{x} \\ &\leq 1 + \ln n \end{aligned} \tag{3.31}$$

3.4 Fonctions classiques

3.4.1 La partie entière

C'est une fonction monotone et croissante. Elle est définie comme suit : pour tout réel x,

1. le plus grand entier inférieur ou égal à x s'écrit $\lfloor x \rfloor$ (ou la partie entière de x) ;
2. le plus petit entier supérieur à x s'écrit $\lceil x \rceil$.

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

Pour tout entier n :

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$$

3.4.2 La fonction exponentielle

Pour tout réel $a \neq 0, m, n$ on a les identités suivantes :

$$\begin{aligned} a^0 &= 1 \\ a^1 &= a \\ a^{-1} &= 1/a \\ (a^m)^n &= (a^n)^m = a^{mn} \\ a^m * a^n &= a^{m+n} \end{aligned}$$

- La fonction a^n est monotone et croissante pour $a \geq 1$. On pose $0^0 = 1$ si nécessaire ;
- comparaison avec un polynôme : $\lim_{n \longrightarrow \infty} \frac{n^b}{a^n} = 0$;
- d'où : $n^b = o(a^n)$, une fonction exponentielle avec une base strictement plus grande que 1 croît plus vite qu'un polynôme quelconque.
- si on s'interesse au réel $2.71828\ldots$ qui est la base e du logarithme népérien, on a :

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

d'où $e^x \geq 1 + x$. L'égalité est vérifiée pour $x = 0$.

- Si $\mid x \mid \leq 1$, alors : $1 + x \leq e^x \leq 1 + x + x^2$.
- On s'intéresse au comportement asymptotique quand $x \longrightarrow 0$, on a $e^x = 1 + x + \Theta(x^2)$, et pour tout x : $\lim_{n \longrightarrow \infty}(1 + \frac{x}{n})^n = e^x$.

3.4.3 La fonction logarithme

Le logarithme d'un nombre dans une base c'est l'exposant qu'il faut donner à la base pour obtenir ce nombre. Par exemple 3 est l'exposant qu'il faut donner à 2 pour obtenir 8, donc $2^3 = 8 \Longrightarrow \log_2 8 = 3$.

- Notations
  1. Logarithme népérien : $\log_e n = \ln n$
  2. Logarithme binaire : $\log_2 n = \frac{\ln n}{\ln 2}$
  3. Exponentiation : $\log^k n = (\log n)^k$
  4. Composition : $\log \log n = \log(\log n)$

- Propriétés de calcul

  Pour tout réels : $a > 0, b > 0, c > 0$ et entier n on a :

  1. $a = b^{\log_b a}$
  2. $\log_c(ab) = \log_c a + \log_c b$
  3. $\log_b a^n = n \log_b a$
  4. $\log_b a = \frac{\log_c a}{\log_c b}$
  5. $\log_b(1/a) = -\log_b a$
  6. $\log_b a = \frac{1}{\log_a b}$
  7. $a^{\log_b n} = n^{\log_b a}$

Ainsi le changement de base d'un logarithme ne modifie la valeur de ce logarithme que d'un facteur constant, qui est négligé dans une notation O.

3.5 Méthodes de résolution des équations de récurrence

Généralement le temps d'exécution d'une fonction récursive s'exprime par une équation de récurrence. L'équation de récurrence décrit une fonction à partir de sa valeur sur des entrées plus petites.

3.5.1 La méthode par substitution

Dans cette méthode, on fait l'hypothèse d'une borne qu'on devine et on démontre par récurrence que cette hypothèse est correcte. La difficulté de cette méthode tient au fait qu'il n'existe pas de méthode générale pour deviner une borne autre que l'expérience de cas similaires.

Exemple 3.5.1 *Démontrer l'existence d'une borne pour l'expression suivante :*

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{3.32}$$

On suppose que la solution c'est $O(n \log n)$*. La méthode consiste à prouver que* $T(n) \leq c * n \log n$ *pour* $c > 0$*.*

On suppose que cette borne est valable pour $\lfloor n/2 \rfloor$*. Donc* $T(\lfloor n/2 \rfloor) \leq c * \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor$*. On remplace* $T(\lfloor n/2 \rfloor)$ *par sa valeur dans l'équation de récurrence. On obtient :*

$$\begin{aligned} T(n) &\leq 2(c * \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq c * n * log(n/2) + n \\ &= c * n \log n - c * n \log 2 + n \\ &= c * n \log n - c * n + n \\ &\leq c * n \log n \quad si\ c \geq 1 \end{aligned} \tag{3.33}$$

On peut imposer que le cas de base de la récurrence c'est $n = 2$ *ou* $n = 3$ *car pour* $n > 3$ *la récurrence ne dépend pas de* $T(1$*. Mais cette valeur de c ne convient pas au cas de base de la récurrence car* $T(n) \leq c * 1 \log 1 = 0$*. Il suffit de choisir* $c \geq 2$*.*

Supposons l'expression suivante à borner :

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \tag{3.34}$$

Ce cas semble différent du précédent mais il parait intuitif que la constante 17 ajouté ne peut pas avoir une influence importante sur la récurrence puis que quand n est grand $\lfloor n/2 \rfloor + 17$ et $\lfloor n/2 \rfloor$ sont proches. On peut donc supposer que $T(n) = O(n \log n)$.

3.5.2 La méthode itérative

Cette méthode transforme la récurrence en une sommation qu'il s'agit de borner.

3.5.3 La méthode générale

Elle donne des bornes pour les récurrences de la forme suivante : $T(n) = aT(n/b) + f(n)$ avec $a \geq 1, b > 1$ et $f(n)$ est une fonction donnée. Elle nécessite d'envisager trois cas

Chapitre 4

Dictionnaire, index et techniques de hachage

4.1 Introduction

Index et dictionnaire sont deux structures de données pour représenter un ensemble de mots et rechercher si un mot appartient ou non à la structure. La différence fondamentale entre ces deux structures c'est les données conservées dans l'une ou l'autre de ces structures, la taille de la structure et la complexité des opérations possibles.

4.2 Dictionnaire

Un dictionnaire est un ensemble de mots qui supporte uniquement les trois opérations suivantes :

1. rechercher un mot
2. insérer un mot
3. supprimer un mot

L'opération de recherche est considérée comme la plus importante. C'est l'unique opération si le dictionnaire est statique, s'il est dynamique, le nombre de consultations (recherche de mots) est généralement très supérieur au nombres d'ajouts et de suppressions de mots, d'où son importance.

Remarque 4.2.1 *De nombreuses applications informatiques font appel à des structures de données du type "dictionnaire". Par exemple en compilation, la gestion de la table des symboles dans le programme consiste à ma-*

nipuler cette table comme un dictionnaire où un symbole est un identificateur dans le programme.

L'implémentation des dictionnaires à l'aide du *hachage* est très efficace. Ces opérations ne nécessitent en moyenne qu'un temps en $O(1)$.
Plus formellement :

Définition 4.2.2 *A tout ensemble de mots fini* $E = \{m_1, m_2, ..., m_x\}$*, on associe une structure* $Dictionnaire(E)$ *qui, étant donné un mot* m *permet de vérifier si oui ou non* m *appartient au dictionnaire. On peut décider de retourner la position* $i, i = \{1, 2, ...x\}$ *de* m *dans le dictionnaire s'il existe, et* 0 *sinon.*

4.3 Index

Un index est une structure de données permettant de représenter les occurrences (positions) d'un ensemble de mots d'un texte ou d'un ensemble de textes.

Rechercher un mot dans le texte consiste à s'intéresser à retrouver, soit sa première occurrence, soit toutes ses occurrences, soit ses occurrences à partir d'une position donnée du texte, soit dans un intervalle de positions donné, etc.

L'index est dynamique si on peut ajouter de nouveaux mots (ou textes) à l'index actuel. Les opérations de base dans un dictionnaire sont généralisées dans la structure d'index. Elle se basent aussi sur les techniques du hachage pour une implémentation plus efficace.

4.3.1 Définition

Définition 4.3.1 *Soit* $E = \{T_1, T_2, ...T_x\}$ *un ensemble de* x *textes. On définit l'index de* E *par l'ensemble suivant :*
$Index(E) = \{(m, p, i) |$ m *est un mot ou un facteur dans au moins un* $T_i, (i = 1, \ldots, x)$, *et* p *une position de* m *dans un* $T_i \in E\}$

Donc si $E = \{T_1, T_2, ...T_x\}$ est un ensemble de x textes, l'index de ces x textes est un ensemble de mots, tel que chaque mot a une occurrence dans au moins un $T_i (i = 1, \ldots, x)$, et p est la position de cette occurrence.

4.3.2 Opérations

- **La recherche** : "Trouver toutes les positions d'un mot dans un texte"
  La principale opération qu'on veut pouvoir appliquer sur un index est l'opération suivante : Étant donné un mot m, trouver l'ensemble de toutes les positions de m dans E, autrement dit,
  " trouver l'ensemble des éléments (p, i) tels que $\{(m, p, i) \in Index(E)\}$".
- **La mise à jour** : "Ajouter ou supprimer un texte à l'ensemble des textes"
  Dans un index dynamique on veut de plus pouvoir ajouter ou supprimer un texte T à l'ensemble E sans reconstruire l'index depuis le début (incrémentalement). Si l'ensemble des mots ne change pas, l'index est statique, donc les opérations de mise à jour ne sont plus nécessaires.

Les sections qui suivent introduisent les notions de base du hachage, la généralisation de cette technique en termes des filtres de Bloom, ainsi que les applications de ces derniers dans un domaine de recherche très dynamique : les réseaux.

4.4 Le hachage

4.4.1 Définitions

Définition 4.4.1 ***Table de hachage***

Une table de hachage *est une structure de données qui permet d'implémenter un dictionnaire. Soit T une table de données (un dictionnaire) et x un élément de T (ou mot du dictionnaire). Une table de hachage permet d'associer une clé d'accès $f(x)$ à chaque élément x de T.*

La recherche d'une clé dans une table de hachage est au pire en $\Theta(n)$, n étant le nombre d'entrées de la table. En pratique on arrive à des algorithmes de recherche qui sont en moyenne en $O(1)$ pour un élément qui existe dans la table.

La table de hachage généralise la notion de tableau classique. L'accès direct à un élément i du tableau se fait en $O(1)$. Cette généralisation est rendue possible grâce à la notion de *clé*.

Définition 4.4.2 ***Fonction de hachage***

La clé, position de l'élément dans la table T, se calcule à l'aide d'une fonction f, dite fonction de hachage. Cette fonction prend en entrée un élément

x de T et fournit en sortie un entier (dans un intervalle donné, par exemple $[0, m]$). La fonction f est construite de manière à ce que les entiers qu'elle fournit soient uniformément distribués dans l'intervalle. Ces entiers sont les indices du tableau T.

Les tables de hachage sont très utiles lorsque le nombre de données effectivement stockées est très petit par rapport au nombre de données possibles. La table de hachage utilise un tableau de taille proportionnelle au nombre de données effectivement stockées. Au lieu d'utiliser la donnée directement comme indice du tableau, cet indice est calculé à partir de la donnée pour retrouver un élément rapidement, sans parcourir le tableau. Cette recherche est en moyenne en $O(1)$.

Soit T un tableau de n entiers tel que $1 \leq T[i] \leq n$. Un algorithme en $O(1)$ consiste à placer les éléments tels que $T[i] = i$.

Exemple 4.4.3 *Soit T le tableau suivant :*

7	*5*	*4*	*6*	*1*	*8*	*2*	*3*

La recherche dans ce tableau est en $O(n)$. Mais si on constate que tous les $T[i] \leq 8$, on peut obtenir une recherche en $O(1)$ avec le tableau suivant :

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*

Mais si n est grand et le nombre de clés présentes dans T est m avec $m \ll n$, on ne peut plus adopter cette solution, on utilise les tables de hachage. C'est le cas d'un dictionnaire de la langue. L'alphabet est $\Sigma = \{a, b, c, d, \ldots, y, z\}$, $|\Sigma| = 26$, le nombre de mots possibles obtenus par combinaisons et permutations des caractères est très grand par rapport au nombre réel des mots d'un dictionnaire. L'arrangement de r lettres parmi 26, donne A_{26}^r mots possibles, soit $\frac{26!}{(26-r)!}$. Si r tend vers 26 (le mot le plus long), ce nombre tend vers 26! *mots possibles*[1].

Définition 4.4.4 ***Collision***

Si n est la taille de T toute clé qui dépasse n sera calculée modulo n. La fonction f n'est donc pas injective : deux éléments différents x_1 et x_2 de T peuvent avoir la même clé $f(x_1) = f(x_2)$. On dit qu'il y a collision.

1. 10 ! = 3 628 800. Rappelons que dans le dictionnaire Robert 2010, il y a 60.000 mots.

Remarque 4.4.5 *Une bonne fonction de hachage doit minimiser le nombre de collisions.*

Exemple 4.4.6 *Soit* $Z = \{11, 59, 32, 44, 31, 26, 19\}$ *un ensemble d'entiers positifs et* T *un tableau de taille 10. On range les élément de* Z *dans* T*, en prenant par exemple la fonction de hachage* $f(x) = z_i$ *modulo* 10.

z_i	*11*	*59*	*32*	*44*	*31*	*26*	*19*
$h(z_i)$	*1*	*9*	*2*	*4*	*1*	*6*	*9*

Il y a collision entre 11 et 31 et aussi entre 59 et 19.

Un algorithme complet de hachage consiste en une fonction de hachage et une méthode de résolution des collisions.

Remarque 4.4.7 *Les éléments sont à la fois insérés et recherchés dans la table en utilisant la fonction de hachage.*

Il y a deux grandes classes distinctes de méthodes de résolution de collisions.

4.4.2 Hachage par chaînage

Dans cette méthode, les éléments en collision sont chaînés dans une liste. Une fois la clé trouvée, la recherche est séquentielle dans la liste. Elle est linéaire en la taille de cette liste. L'avantage de cette méthode est la simplicité de l'ajout et de la suppression. L'insertion est en O(1) dans le pire des cas (on insère en tête de liste). La suppression est en O(1) si les listes sont doublement chaînées. Si les listes sont simplement chaînées, il faut parcourir pour garder l'adresse du précédent, donc au pire c'est comme la recherche. Au lieu d'une liste, on peut utiliser un arbre binaire équilibré ou un tableau dynamique. Il existe plusieurs autres méthodes de résolution des collisions.

Définition 4.4.8 *La complexité en moyenne*

Soit A *un algorithme et* d *une donnée, on note* $cout_A(d)$ *le nombre d'opérations élémentaires pour traiter la donnée* d*. La complexité en moyenne de l'algorithme* A *sur des données de taille* n *est définie par l'expression suivante :*

$$\sum_{d \in D_n} p(d) * cout_A(d)$$

$p(d)$ est la probabilité pour que la clé d soit tirée.

Définition 4.4.9 *Le facteur de remplissage*
Soit T une table de hachage de m cases, contenant n éléments. On définit le facteur de remplissage α par le rapport n/m qui représente le nombre moyen d'éléments stockés dans une chaîne.

Pour l'analyse du hachage, on suppose que α reste fixe quand n et m tendent vers l'infini. Le cas pire du hachage par chaînage arrive quand les n clés sont chaînées dans la même liste (les n clés ont la même valeur de fonction de hachage). La recherche est en $\Theta(n)$, (il faut parcourir toute la liste des n clés) plus le temps de calculer les valeurs de la fonction.

La complexité en moyenne du hachage dépend de la manière dont la fonction répartit l'ensemble des clés entre les m valeurs possibles. On suppose que chaque élément à la même probabilité d'être haché dans l'une des cases, indépendamment où les autres éléments ont été hachés (une fonction de hachage uniforme simple).

Remarque 4.4.10 *Une bonne fonction de hachage vérifie l'hypothèse de hachage uniforme simple. Si chaque clé est tirée indépendamment de l'ensemble de l'univers des clés selon une loi de probabilité P ($P(k)$ est la probabilité d'avoir la clé k par la fonction, l'hypothèse de hachage simple uniforme signifie que :*

$\sum_{k:f(k)=j} P(k) = \frac{1}{m}$ *pour* $j = 0, 1, \ldots, m-1$.

Toute la difficulté est dans la détermination de P souvent inconnue.

Théorème 4.4.11 *Dans ce type de hachage, une recherche qui échoue prend en moyenne un temps en $\Theta(1+\alpha)$*

En effet, α est la longueur moyenne des listes. Une recherche qui échoue parcourt une des m listes qui est de longueur α, plus le temps de calcul de la fonction de hachage.

Théorème 4.4.12 *Dans ce type de hachage, une recherche qui réussit prend en moyenne un temps en $\Theta(1+\alpha)$*

4.4.3 Adressage ouvert

Dans cette seconde méthode dite " adressage ouvert " (open-addressing) tous les éléments sont conservés dans la table de hachage elle-même. Chaque entrée de la table contient soit une clé, soit NUL. Si k est une clé, on vérifie si $f(k)$ est dans la table ou non, si on ne le trouve pas, on calcule un nouvel indice basé sur la clé, c'est-à-dire *re-hache* dans la table jusqu'à trouver la clé ou NUL si l'élément n'appartient pas à la table.

On peut aussi chaîner dans cette même table les éléments ayant même valeur de fonction de hachage (en collision), mais cette solution est moins intéressante puisque la recherche implique une partie de recherche séquentielle en suivant les pointeurs.

Dans ce type de hachage, la table peut se remplir entièrement et on ne peut plus insérer une nouvelle clé. L facteur de remplissage ne doit jamais être supérieur à 1.

Théorème 4.4.13 *Soit une table de hachage (uniforme simple) en adressage ouvert avec un facteur de remplissage $\alpha = n/m < 1$. Le nombre d'éléments examinés pour une recherche qui échoue vaut au plus $1/(1-\alpha)$.*

Théorème 4.4.14 *Soit une table de hachage (uniforme simple) en adressage ouvert avec un facteur de remplissage $\alpha = n/m < 1$. Le nombre d'éléments examinés pour une recherche qui réussit vaut au plus $\frac{1}{\alpha} * \ln(\frac{1}{1-\alpha})$.*

4.5 Types de hachage

Il y a plusieurs types de hachage dont l'efficacité dépend des contraintes de leur application.

1. **Hachage statique**

   Dans le hachage statique les données sont stockées dans des tables de taille fixe. Un espace mémoire statique est alloué pour contenir la table de hachage. Si l'espace alloué est trop grand on risque une perte d'espace inutile, s'il est insuffisant il faut restructurer les tables.

2. **Hachage dynamique**

   Il permet d'étendre le hachage statique pour s'adapter dynamiquement à des tailles de tables (ou fichiers) de données croissantes ou décroissantes sans pénalité de coût.

3. **Hachage parfait**

   C'est un hachage dans lequel il n'y a pas de collision ; sa fonction est injective.

   Pour atteindre la solution idéale d'un nombre nul de collisions, on se base sur l'idée de rendre le hachage aléatoire.

4. **Table de hachage distribué** [2]

   C'est une structure de données qui permet d'identifier et de retrouver une information dans un système réparti, comme les réseaux P2P. Chaque nœud du reseau gère une partie de la table de hachage, il est responsable d'un certain intervalle de clés.

   L'utilisation d'un hachage réparti implique qu'il n'y a pas de connaissance globale centralisée de l'ensemble des données. Les ressources sont uniformément réparties entre les nœuds.

   L'algorithme de base d'un tel système consiste à retrouver le nœud responsable d'une clé donnée. Les clés et les nœuds reçoivent un identificateur de m bits : l'identificateur nœud provient du hachage de son adresse IP, l'identificateur clé provient du hachage de la clé. La fonction de hachage doit avoir de bonnes propriétés (minimisation des collisions).

4.6 Applications

L'étude des algorithmes et techniques de recherche de motifs dans le contexte du filtrage du trafic dans les réseaux semble être une voie de recherche prometteuse, en particuliers les méthodes basées sur l'utilisation des filtres de Bloom. Cette structure de données basée sur le hachage probabiliste s'est avérée intéressante dans plusieurs applications réseaux comme le traitement du routage, le contrôle du réseau, la détection d'intrusion (IDS) ou la prévention d'intrusion (IPS) pour la sécurité des réseau. L'intérêt des filtres de Bloom vient du fait qu'ils permettent d'obtenir des algorithmes très rapides avec un coût d'implémentation hardware réduit. Les applications possibles peuvent concerner différents types de réseaux, comme par exemple : les réseaux à haut débit, les réseaux mobiles, les réseaux de capteurs etc.

2. DHT : Distributed hash table

Chapitre 5

Les classes de problèmes

5.1 But de la classification

La complexité des algorithmes permet de classer les problèmes en fonction de la difficulté à les résoudre. On peut déjà comprendre intuitivement cette notion de "difficulté" en comparant les deux problèmes suivants :

1. soit n un entier donné, vérifier si n est pair ou impair (il est pair si le dernier chiffre vaut $\{0, 2, 4, 6, 8\}$ et impair sinon)
2. soit n un entier donné, vérifier si n est un nombre parfait ou non (il est parfait s'il est égal à la somme de ses diviseurs et ne l'est pas sinon).

Une complexité est acceptable si elle est polynômiale.

Définition 5.1.1 ***Algorithme polynômial***
Un algorithme est dit en temps polynômial si, pour tout n, n étant la taille des données, l'algorithme s'exécute en temps borné par un polynôme $f(n) = C * n^k$ opérations élémentaires (les constantes C et k sont indépendantes de n). Par exemple $f(n) = 5n^2$.

Définition 5.1.2 ***Problème polynômial***
On dit qu'un problème p est polynômial ou encore que p est dans la classe P si et seulement si il existe un algorithme pour le résoudre qui s'exécute en temps polynômial.

En informatique on considère que les algorithmes polynômiaux sont les seuls à pouvoir être utilisés pour de grandes tailles des données, indépendamment de la puissance de la machine. Ainsi, les problèmes de coût exponentiel ou plus sont supposés insolubles en pratique dans le cas général. Ceci veut dire qu'il est possible qu'ils puissent être résolus dans des cas particuliers. Parmi les problèmes de coût exponentiel, certains ont la propriété

suivante : on peut vérifier leur réponse en temps polynomial. On dit que ces problèmes sont NP.

Définition 5.1.3 ***Problème NP***
Un problème p est dans la classe NP si et seulement si il existe un algorithme qui vérifie si une réponse au problème p est correcte ou non en temps polynomial.

En général, vérifier une réponse est toujours moins coûteux que la calculer. Par exemple vérifier qu'une liste de longueur n est ordonnée est moins coûteux (en O(n)) que la trier (en O(nlogn)).

Ainsi la classe NP contient l'ensemble des problèmes dont la vérification est polynômial mais dont la résolution ne l'est pas obligatoirement

Remarque 5.1.4 *La définition précédente implique l'inclusion $P \subseteq NP$. Mais on n'a pas démontré l'existence d'un problème qui est dans NP mais pas dans P, autrement dit on ne sait pas si $P \neq NP$. Cette question est l'un des problèmes ouverts les plus importants de la recherche en informatique fondamentale aujourd'hui.*

Exemple 5.1.5 *Le problème de la satisfiabilité d'une formule du calcul propositionnel est dans NP*

Il est facile de vérifier, en temps polynomial qu'une formule propositionnelle vaut vrai étant donnée une affectation de valeurs de vérité à ses variables. Mais jusqu'à aujourd'hui on ne connaît pas d'algorithme polynomial qui donne les valeurs de vérité aux n variables de la formule. Tous les algorithmes connus sont des variantes de l'énumération des 2^n affectations possibles, donc des algorithmes exponentiels.

5.2 Les problèmes P et NP

Définition 5.2.1 ***La réductibilité***
Un problème p_1 peut être ramené (ou réduit) à un problème p_2 si une instance quelconque de p_1 peut être traduite comme une instance de p_2 et la solution de p_2 sera aussi solution de p_1. On note cette réduction : $p_1 \preceq p_2$.

Du point de vue de la complexité, cette définition permet de déduire les deux conclusions suivantes :

- que le coût de résolution du problème p_1 est au plus égal au coût de la transformation plus le coût minimum connu de la résolution de p_2. Donc si p_2 est dans la classe P (se résout en temps polynomial) et si la transformation est aussi dans la classe P alors p_1 est aussi dans la classe P.
- mais si le coût minimum de résolution de p_1 est connu, alors le coût de p_2 plus celui de la transformation ne peut pas être moindre. Donc si p_1 a un coût exponentiel alors soit la transformation a un coût exponentiel soit le coût de p_2 a un coût exponentiel (soit les deux).

Notons qu'il est nécessaire que la fonction de transformation soit polynomiale sinon les deux conclusions précédentes ne peuvent pas être faites.

5.2.1 Propriétés

- Deux problèmes se réduisant mutuellement l'un en l'autre avec un coût polynomial sont soit tous les deux de coût polynomial soit tous les deux de coût exponentiel. Tout progrès pour l'un est valable pour l'autre, ce qui veut dire que si on découvre un algorithme polynomial pour l'un (alors qu'on ne lui connaissait que des algorithmes exponentiels) alors il s'applique immédiatement à l'autre.
- Il existe une famille de problèmes de la classe NP vers lesquels tous les problèmes de NP peuvent se réduire. On appelle cette famille, les problèmes NP-complets. Ce sont les problèmes les plus coûteux, les plus difficiles à résoudre. Si on prouve qu'un problème NP-complet n'est pas dans la classe P alors la preuve est valable pour tous les autres. Et si on montre l'inverse, autrement dit l'existence d'un problème de la classe NP-complet qui admet un algorithme polynomial, alors le résultat sera valable pour tous les autres de cette classe.
- La relation de réductibilté est réflexive et transitive.

Définition 5.2.2 ***Problème NP-complet***

On dit qu'un problème p est NP-complet si et seulement si :

1. $p \in NP$ *et*
2. $\forall q \in NP \quad q \preceq p$

Une problème NP-complet est un problème pour lequel on ne connaît pas d'algorithme "réalisable" c'est-à-dire polynomial. Le problème le plus célèbre est le problème du voyageur de commerce. L'ensemble des problèmes NP-complets ont les propriétés suivantes :

- *Si on trouve un algorithme efficace pour un problème NP complet alors il existe des algorithmes efficaces pour tous,*
- *Personne n'a jamais trouvé un algoritrhme efficace pour un problème NP-complet,*
- *personne n'a jamais prouvé qu'il ne peut pas exister d'algorithme efficace pour un problème NP-complet particulier.*

5.3 Quelques problèmes classiques NP-complets

5.3.1 Le problème du sac à dos

A votre droite, un sac à dos vide. A votre gauche, m objets, de poids respectifs $a_1, \ldots, a_m$. Quels objets doit on mettre dans le sac à dos pour obtenir un poids total égal à k ?

Il s'agit là d'un problème très difficile, à la fois en pratique (il est insoluble, même avec un ordinateur, pour de grandes valeurs de m), et en théorie (il fait partie des problèmes NP-complets).

5.3.2 Le problème du voyageur de commerce

Ce voyageur doit visiter n villes pour vendre sa marchandise. Il existe entre ces villes un certain nombre de routes. Existe-t-il un trajet permettant de visiter toutes les villes en moins de b kilomètres. C'est un problème difficile, et on ne sait pas s'il existe un algorithme polynômial pour le résoudre. En revanche, si l'on donne un trajet, il est quasi-immédiat de vérifier si ce trajet fait moins de b kilomètres. Le problème est dans NP.

5.4 Problèmes exponentiels

Les problèmes de complexité exponentielle sont des problèmes ouverts. C'est les challenge de la recherche. La fonction de complexité est de la forme $f(n) = C^n, c > 1$. Par exemple si cette fonction est une valeurs de temps exprimée en nanosecondes avec $C = 2$ et $n = 100$ alors $2^{100} \approx 10^{30}$ elle représente un temps de $3 * 10^{11}$ siècles ! Faire le même calcul avec $n = 1000$.

Prouver qu'un problème est NP-complet (par réduction) c'est prouver qu'il est intraitable et donc il est inutile de chercher sa solution. On peut s'intéresser alors à chercher une solution approchée.

5.5 Problèmes de décision et problèmes d'optimisation

La théorie de la NP-complétude s'interesse aux problèmes de décision (un problème admet oui/non une solution). Mais parfois nous sommes face à un problème d'optimisation.

Dans un problème d'optimisation, chaque solution réalisable a une valeur qui lui est associée et on veut trouver la solution réalisable qui a la valeur optimale.

Par exemple, dans le *problème du plus court chemin* dans un graphe on a :

- un graphe orienté G ;
- des sommets u et v
- et on veut trouver le chemin de u à v qui utilise le moins d'arcs.

Un problème de décision est un problème pour lequel la réponse est **oui** ou **non** (0 ou 1).

La NP- complétude s'applique directement aux problèmes de décision, mais il y a une relation entre problème d'optimisation et problème de décision.

On peut convertir un problème d'optimisation en un problème de décision en imposant une borne à la valeur à optimiser.

Par exemple : le problème de décision associé au problème d'optimisation du plus court chemin c'est le problème *Chemin* suivant :

- Soit G un graphe orienté ;
- soit u et v deux sommets ;
- soit k un entier
- existe t-il un chemin de u à v ayant au plus k arcs ?

On utilise cette relation lorsqu'on veut montrer qu'un problème d'optimisation est NP-complet. Il est plus facile de le montrer sur le problème de décision qui lui est associé. Cette réduction implique que si le problème d'optimisation est facile alors le problème de décision sera aussi facile. Et si on prouve que le problème de décision est difficile alors le problème d'optimisation le sera aussi.

Chapitre 6

Stratégies de résolutions des problèmes

1. Diviser pour régner
2. Programmation dynamique
3. Algorithmes gloutons