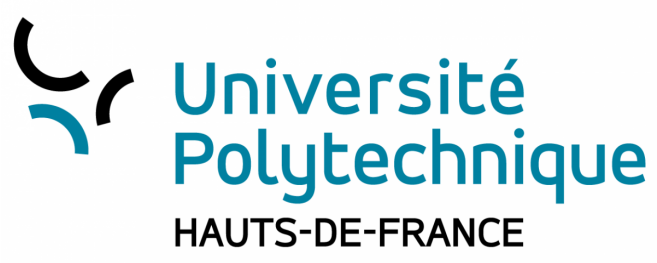


Bases de la Complexité *

S3INE3 : Licence 3 Informatique

R. Mandiau

*Version : Mars 2020

Table des matières

1 Introd Gen.

1.1 Motivations

Répondre à des questions essentielles pour l'Informatique

- En quoi l'étude des **algorithmes** est-elle utile ?
- Comment **analyser** rigoureusement les algorithmes ?
- Quel est son **Rôle/Intérêt** par rapport aux autres technologies informatiques (Base de Données, Web, etc.) ?
- **Évolution technologique vs Analyse théorique du comportement des Algorithmes**
- Complexité Temporelle et/ou Spatiale : **un enjeu crucial !**
  - Temps de calcul : Nombre d'opérations à effectuer
  - Taille mémoire : Quantité de mémoire utilisée
  - *Charge de communication : Nombre et Taille des messages émis et reçus*

Quelques justifications classiques

- Performances (grande taille des données, nombre très élevé d'appels, contraintes temporelles/ temps-réel)
- Applications typiques (systèmes de robotique et de véhicule autonome, fouille de données, traitement de données massives – Big Data)

Dans ce cours ...
Complexité Temporelle des algorithmes

1.2 Quelques applications ...

ADN Humain

Identifier les 100000 *gènes de l'ADN humain :* Déterminer les séquences des 3 milliards de paires de bases chimiques qui constituent l'ADN humain, de stocker/de rechercher ces informations dans des Bases de Données ?

(Hypothèse : 1 ordinateur = 1μs/opération = 10^{-6}secondes/oper.)

- *Comparer deux séquences ADN :* Algorithme qui compare élément par élément

  Temps estimé : 3000 secondes ($3 \times 10^9 \times 10^{-6}$)
- *Rechercher un code ADN* dans une Base de Données de 1 Milliard de personnes :

  Temps estimé : 3×10^{12} sec. = 96000 années
- Enjeux actuels : Déterminer les *marqueurs* caractéristiques.

Les Jeux

Sudoku (grilles 9×9) - (étude de Bertram Felgenhauer en 2005) [16]

— Nombre de grilles existantes à générer : $6'670'903'752'021'072'936'960 \approx 6.67.10^{21}$

Temps estimé : $6.67.10^{15}$ sec = $2.12.10^{8}$ années

— Nombre de grilles existantes avec prise en compte des rotations, symétries, etc. : $5'472'730'538$

Temps estimé : $5.47.10^{3}$ sec. = 1.52 jours

Échecs

— Génération des différents nœuds : estimation à 10^{120}

Temps estimé : 10^{109} millénaires

(Hypothèse : création d'une grille en 1μs)

Systèmes d'Aide à la Décision (à base d'IA/RO)

— Industrie/Commerce : Affecter et/ou allouer de ressources limitées, maximiser des profits, etc.

— Transports : Recherche du chemin le plus court pour des camions et/ou SNCF, régulation de bus, trafic routier, matières dangereuses, etc.

— Étude de trafic routier (coût : dépasse le million d'euros)

— Plusieurs semaines pour des outils macroscopiques (étude dynamique des flux de véhicules)

— Enjeux actuels : Développement d'Outil microscopique proche du temps réel.

Internet

— *Nombre de pages :* 19.2 milliards de page (recensés par Yahoo en 2005)

— *Info recensés par Netcraft en* 2007 *:*

  — *Nombre de sites Web distincts :* 108810358

  — *Nombre moyen de pages par site :* 293

  — Soit un nombre de pages estimé à 29.7 milliards

— Chiffre d'affaires : 2 Milliards d'euros en 2006

— Nombre d'internautes en France : 27.21 millions (Dec. 2005)

source : http ://www.declik-interactive.com/chiffrescles.htm (accessible Juillet 2007)

Internet : Temps de recherche exhaustive ?

— Recherche exhaustive d'une page spécifique ?
Temps estimé : 29.7 milliards $\times 1\mu s$ = 29700 secondes = 8.2 heures (aucune prise en compte du temps mis pour la recherche de l'information sur les différents nœuds du réseau)

— Recherche d'un mot-clé sur les différentes pages ? (Hypoth : 500 mots par page)
Temps estimé : 5 mois (avec un algo de recherche séquentielle)

— Enjeux actuels : Recherche les routes optimales pour l'acheminement des données (routage), utilisation d'un (ou des) moteurs de recherche pour trouver rapidement les pages (indexation des pages)

(Hypothèse : création d'une grille en $1\mu s$)

Internet : Stockage d'une grande bibliothèque virtuelle ?

— Stockage de toutes les pages Web

— Hypothèses
  — 500 mots par page ;
  — 1 mot est en moyenne de 5 caractères (1 caractère = 1 octet)

— Estimation : D'où 29.7 milliards $\times$2500 car.= 74.25 To ($1Tera-Octet = 10^{12}$ octets)

Surveillance électronique

Le problème de stockage et de la recherche d'information se pose aussi dans la surveillance électronique :

— Évaluation du nombre d'images à mémoriser pour une agglomération (Paris ou Londres) ?

— Temps de Recherche d'un individu si le stockage est techniquement possible ?

Pour info : Le niveau de compression (actuellement)

— Compression de textes : gain en moyenne de 50% (décompression doit donner le même texte)

— Compression d'images statiques : gain en moyenne de 80% (décompression tolérance à une dégradation)

— Compression d'images d'un film : gain en moyenne de 95%

Estimation personnelle : Stockage des Images sur un an ?

— 24 images par seconde

— 5000 caméras à Londres !

— 1 image est constituée de 800×800 pixels (1 bit = 1 pixel)

— Stockage des images sur un an

— Taux de compression : gain de 95%

— $24 \times (3600 \times 24 \times 365 jours) \times (800 \times 800) \times (1 - 95\%)$ bits
Soit $1.5 \times 10^4 To$

Complexité du Vivant : Téléportation d'un individu

étude extraite du livre de J.P. Delahaye (L'Intelligence et le calcul, Belin Eds.) [3]

- Quantité d'information pour décrire un être humain : 10^{32} bits
- Compression des données : 10^{20} bits
- Réduction des informations manipulées : 10^{15} bits
  - Redondances informationnelles (génome identique dans toutes les cellules)
  - Conservation des informations pertinentes (qu'importe alors la longueur exacte des cheveux, la position précise de chaque pore de la peau, etc.)
- On peut considérer que :
  - Vu la taille du génome (moins de 10^{10} bits d'informations),
  - Vu le vocabulaire qu'une personne connaît (50000 mots, correspond à 10^{7} bits)
  - Vu le nombre de souvenirs qu'une personne possède (un livre de 100000 pages pour les consigner, cela représente moins de 10^{9} bits.
- Donc : Il semble que 10^{15} bits de programmes caractérisent un individu (certains chercheurs pensent que 10^{12} ou 10^{10} peut être suffisant).

  Stockage des données d'un corps humain (*The Visible human project*, Texas) en 3 Dimensions aux millimètres prés (cadavre congelé qui a été découpé en fines tranches) : 10^{11} bits

10^{12} bits est équivalent à une bibliothèque de 100000 livres de 200 pages

2 Notions

2.1 Introduction

Principaux concepts

— L'algorithme s'arrête-t-il ? (preuve de terminaison)

— L'algorithme donne-t-il une solution (preuve de correction)

— L'algorithme donne une solution correcte dans un temps acceptable ? (problème de complexité)

Algorithme : Définition informelle

— Procédure de calcul bien définie qui prend en entrée une valeur (ou un ensemble de valeurs) et qui donne en sortie une valeur (ou un ensemble de valeurs)

— Outil permettant de résoudre un problème de calcul bien spécifié. L'énoncé du problème spécifie la relation désirée entre l'entrée et la sortie.

Exemple 1 : Tri d'une suite de nombres

— entrée : Suite de n nombres $(a_1, a_2, \cdots, a_n)$

— sortie : Permutations de la suite donnée en entrée telle que $(a'_1, a'_2, \cdots, a'_n)$ avec $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

2.2 Quelques Définitions sur les algorithmes : Rappel

Pour ce rappel, nous allons présenter très rapidement quelques définitions utiles sur les algorithmes, à savoir (i) un algorithme correct vs incorrect, (ii) un algorithme décidable vs indécidable, et (iii) un algorithme efficace.

Propriétés d'un algorithme

Definition 1. (i) Algorithme correct vs incorrect

— Un algorithme est dit correct si, pour chaque instance en entrée, il se termine en produisant la bonne sortie.

— Un algorithme incorrect risque de ne se pas se terminer pour certaines instances en entrée, voire de se terminer sur une réponse autre que celle désirée.

Exemple 2 : Un algorithme incorrect peut s'avérer utile si son taux d'erreur est susceptible d'être contrôlé (par exemple, les algorithme des grands nombres premiers)

Definition 2. (ii) Algorithme décidable vs indécidable

— Un algorithme est dit décidable si, pour chaque instance en entrée, il se termine en produisant la bonne sortie en un temps fini ;

— Un algorithme indécidable s'il risque de ne se pas se terminer pour certaines instances en entrée.

Exemple 3 : Deux problèmes connus indécidables

— en Intelligence Artificielle, Logique des prédicats

— Le dixième problème de Hilbert est indécidable : Il consiste à déterminer si l'équation $p(x_1, x_2, \cdots, x_n) = 0$ où $p(x_1, x_2, \cdots, x_n)$ est un polynôme à coefficients entiers a une solution dans le domaine des entiers. Par exemple, $x^2 - 4$ a une solution entière, alors que $x^2 - 2$ ne le permet pas en solution entière.

(iii) Définition d'un Algorithme efficace

Definition 3. — Durée que met un algorithme à produire ses résultats (Mesure d'efficacité d'un algorithme)

— Il existe des problèmes pour lesquels l'on ne connaît aucune solution efficace (Un sous-ensemble est appelé des problèmes **NP – complets**)

Remarque : Il n'existe pas de réponse absolue sur la signification d'un algorithme considéré comme efficace. Une complexité exponentielle en c^n avec $c > 1$ est presque toujours excessive. Considérons par exemple 2^n : pour $n = 100$ (taille de problème modeste), nous obtenons un nombre $2^{100} \approx 10^{30}$ largement excessif. Admettons donc qu'une complexité exponentielle est excessive. Cela ne veut pas dire qu'une fonction de complexité croissant moins vite que l'exponentielle est acceptable : en général, ce n'est pas vrai. Comparer à une exponentielle, il est tentant d'y fixer la limite d'une complexité non exponentielle comme acceptable. Nous considérons comme acceptable une complexité polynomiale en n^k pour un k fixé. Il est clair qu'un algorithme de complexité n^{100} n'est pas très efficace. En pratique, quand un algorithme est polynomial, son polynôme est de degré peu élevé (inférieur à 5). Les algorithmes polynomiaux existants sont donc souvent efficaces ; mais pour certains problèmes, il n'existe pas d'algorithmes efficaces.

Pour un problème donné, deux algorithmes peuvent différer en terme d'efficacité

Exemple 4 : Tri d'une liste de n éléments (c_1 et c_2 des constantes)

— Tri par insertion prend un temps égal à $c_1 \times n^2$

— Tri par fusion prend un temps égal à $c_2 \times n \times \log_2(n)$

c_1 et c_2 dépendent de différentes caractéristiques (ordinateur, programmeur, environnement et langage informatique, etc.).

En général :

— Tri par insertion plus rapide pour n petit

— Tri par fusion plus rapide pour n suffisamment grand

Illustration de l'efficacité de deux algorithmes de Tri

Exemple 5 : Supposons les hypothèses proposées dans le Tableau 1. Ce tableau introduit les principales caractéristiques de l'efficacité de mon algorithme, à savoir la méthode de tri utilisée, la puissance de calcul de l'ordinateur, le niveau du programmeur. Celles-ci donnent un coût résultant.

TABLE 1 – Caractéristiques impactant l'efficacité

Nom	Méthode de Tri	Puissance de calcul	Niveau du programmeur	Code résultant
Ordinateur A (rapide)	Insertion	1 milliard	le *Meilleur*	$2 \times n^2$ (i.e. $c_1 = 2$)
Ordinateur B (lent)	Fusion	10 millions	le *Moins bon*	$50 \times n \times \log_2(n)$ (i.e. $c_2 = 50$)

Note : La puissance de calcul est évaluée en nombre d'instructions par seconde : A est 100 fois plus rapide que B en terme de puissance *brute* de calcul (Tableau 2).

TABLE 2 – Illustration en fonction du nombre d'éléments

Nom	1 million d'éléments	10 millions d'éléments
Ordinateur A	2000 secondes	2.3 jours
Ordinateur B	$\approx$ 100 secondes	20 minutes

Démonstration. Pour rappel :

- $\frac{2\times(10^6)^2}{10^9}$ = 2000 secondes
- $\frac{50\times10^6 log_2(10^6)}{10^7} \approx$ 100 secondes

□

2.3 Temps & Mémoire

2.3.1 Complexité Temps

Analyse de la Complexité Temps d'un Algorithme

1. Opérations fondamentales
   - Nombre de comparaisons
   - Nombre d'additions/multiplications
   - Nombre de permutations

— **Toute Opération primitive** (affectation, comparaison, etc.)

— Les commentaires et les primitives d'E/S ne sont pas considérés comme des opérations fondamentales

2. Taille de l'entrée (Nombre d'éléments constituant l'entrée) : dépend du problème étudié
3. Temps d'exécution

Exemple 6 : algorithme de Tri

1. Opérations fondamentales : nombre de comparaisons, nombre de permutations, toute opération primitive, etc.
2. Taille de l'entrée : nombre d'éléments à trier
3. Temps d'exécution
   — Durée différente pour 100 éléments, 1 million d'éléments, 10 millions d'éléments
   — Durée différente pour un même nombre d'éléments (trié ou non), idem pour un type différent des éléments

Algorithme du Tri par insertion

Algorithme 1 : Tri_insertion(A,n)

```
   Entrées : A[1..n] non triée, n entier
   Sorties : A[1..n] triée
1.1 début
1.2   pour j ← 2 à n faire
1.3     cle ← A [j];
1.4     // insere A[j] dans la séquence triée A[1..j - 1]
1.5     i ← j − 1;
1.6     tant que (i > 0) et (A[i]> cle) faire
1.7       A[i + 1] ← A[i];
1.8       i ← i − 1;
1.9     A[i + 1] ← cle;
```

— Algorithme efficace pour un petit nombre d'éléments.

Une première analyse détaillée du tri par insertion ***Remarques :***

— Pour la boucle pour $j = 2$ à n : il y a $(n-1)$ itérations + 1 test de terminaison de boucle

— Commentaires ne sont pas des instructions exécutables (consomment un temps nul) !

— Dans la suite, t_j est le nombre de fois que la boucle TQ se termine normalement

Exercice 1 : **Étude du cas favorable**

Démonstration. Il faut déjà déterminer ce que nous entendons pour le cas favorable. Pour un algorithme quelconque de tri, rappelons que la liste initiale est supposée déjà triée dans le meilleur des cas.

Montrons que la complexité est linéaire dans le cas favorable (Tableau 3). Nous devons essayer d'évaluer le nombre d'itérations pour chaque opération fondamentale : dans notre, toute primitive.

TABLE 3 – Cas Favorable Tri par insertion

		coût	Nbre de fois
	Entrées : $A[1..n]$ non triée, n entier		
	Sorties : $A[1..n]$ triée		
1.1	**début**		
1.2	**pour** $j \leftarrow 2$ à n **faire**	c_1	n
1.3	$cle \leftarrow$ A [j] ;	c_2	$n-1$
1.4	// insère A[j] dans la séquence triée A[1..j - 1]	-	-
1.5	$i \leftarrow j-1$;	c_4	$n-1$
1.6	**Tq** $(i>0)$ et $(A[i]>cle)$	c_5	$\sum_{j=2}^{n} 1$
1.7	$A[i+1] \leftarrow A[i]$;	c_6	0
1.8	$i \leftarrow i-1$;	c_7	0
1.9	$A[i+1] \leftarrow cle$;	c_8	$n-1$

$$\begin{aligned} T(n) &= c_1.n + c_2.(n-1) + c_4.(n-1) + c_5.(n-1) + \\ &\quad + c_8.(n-1) \end{aligned}$$

□

Exercice 2 : Proposer une étude similaire pour le cas défavorable et montrer dans ce cas que le tri par insertion est en complexité quadratique.

Cas Général

	Algorithme	*coût*	*Nbre de fois*
	Entrées : $A[1..n]$ non triée, n entier		
	Sorties : $A[1..n]$ triée		
1.1	**début**		
1.2	**pour** $j \leftarrow 2$ *à* n **faire**	c_1	n
1.3	$cle \leftarrow$ A [j];	c_2	$n-1$
1.4	// insere A[j] dans la seq. triée A[1..j - 1]	-	-
1.5	$i \leftarrow j-1$;	c_4	$n-1$
1.6	**tant que** *(i > 0) et (A[i]> cle)* **faire**	c_5	$\sum_{j=2}^{n} t_j$
1.7	$A[i+1] \leftarrow A[i]$;	c_6	$\sum_{j=2}^{n} (t_j - 1)$
1.8	$i \leftarrow i-1$;	c_7	$\sum_{j=2}^{n} (t_j - 1)$
1.9	$A[i+1] \leftarrow cle$;	c_8	$n-1$

$$T(n) = c_1.n + c_2.(n-1) + c_4.(n-1) + c_5.\sum_{j=2}^{n} t_j + c_6.\sum_{j=2}^{n} (t_j - 1) + c_7.\sum_{j=2}^{n} (t_j - 1) + c_8.(n-1)$$

Complexité Temporelle : Trois cas à étudier

- Cas Favorable (cas optimal) : Tableau est déjà trié
  - pour $j = 2, 3, \cdots, n$: $a[i] > cle$ non vérifié dans TQ) donc : $t_j = 1$ itération
  - $T(n) = (c_1 + c_2 + c_4 + c_5 + c_8).n - (c_2 + c_4 + c_5 + c_8)$
  - Conclusion : Temps d'exécution est une fonction linéaire $T(n) = a.n + b$
- Cas Défavorable (pire des cas) : Tableau est trié dans l'ordre décroissant
  - $t_j = j$ itérations
  - $T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right).n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2}\right).n - (c_2 + c_4 + c_5 + c_8)$
  - Conclusion : Temps d'exécution est une fonction quadratique $T(n) = a.n^2 + b.n + c$
- Cas moyen (souvent presque aussi mauvais que le cas défavorable)
  - Conseil : $t_j = \frac{j}{2}$
  - Résultat à montrer : $T(n)$ est une fonction quadratique

Complexité Temporelle : choix du Cas Défavorable

- Temps d'exécution est une borne supérieure :
  « Certitude de ne jamais rencontrer le pire »
- Pour certaines applications, le cas le plus défavorable survient souvent
- Le cas moyen est souvent proche du cas le plus défavorable

Complexité Temporelle : Hypothèses simplificatrices

— Faciliter l'analyse de la complexité par ignorance des coûts réels de chaque instruction (utilisation des constantes)

— Ordre de grandeur (recherche du terme dominant)

  — Tri par insertion a un temps d'exécution dans le cas le plus défavorable de $\Theta(n^2)$ (lire "theta de n 2")

  — Un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable a un ordre de grandeur inférieur.

  — Comparaison d'algorithmes : $\Theta(n^2)$ s'exécute plus rapidement dans le cas le plus défavorable qu'un algorithme $\Theta(n^3)$ (entrée de taille assez grande)

2.3.2 Complexité Mémoire

Complexité espace mémoire

1. Besoin en espace mémoire fixe (noté S_fixe)
2. Besoin en espace mémoire variable (noté $S\_variable(I)$)
3. Espace mémoire total : $S(I) = S\_fixe + S\_variable(I)$

Exemple 7 :

Fonction Simple(a,b) : entier

Entrées : a, b : entier

-.1 **début**

-.2 Simple $\leftarrow \frac{a+b}{2}$

Hypothèses : 1 entier : 2 octets, 1 réel : 4 octets ;

Complexité espace mémoire (suite)

1. Besoin en espace mémoire fixe (noté S_fixe)
   — Besoins fixes : 3 entiers : $S\_fixe = 6$
2. Besoin en espace mémoire variable (noté $S\_variable(I)$)
   — Besoin Variable : $S\_variable(I) = 0$
3. Espace mémoire total : $S(I) = S\_fixe + S\_variable(I)$
   — $S(I) = 6$ octets (i.e. $S(I) = O(1)$)

Complexité Espace mémoire : Tri par insertion

Variables : i, j, cle, n : entier ; Tableau d'entiers : A[1..n]

1. Pascal : Tableau transmis par valeur (tableau recopié)
   — Besoins fixes : 4 entiers ($S\_fixe = 4$)
   — Besoins variables : Tableau A[1..n] (i.e. $S\_variable(I) = n$)

2. C : Tableau transmis par adresse (1 adresse = 4 octets)
   — Besoins fixes : 4 entiers + 1 adresse ($S\_fixe = 6$)
   — Besoins variables : $S\_variable(I) = 0$
3. Complexité espace mémoire $S(I) = 4 + n$ ou $S(I) = 6$ (en entiers)

2.4 Interprétation

2.4.1 Notions de complexité et fonctions

Interprétation de la notion de Complexité

On définit des « classes canoniques » de complexité :

— $f(n) = 1$: complexité constante
— $f(n) = \log(n)$: complexité logarithmique
— $f(n) = n$: complexité linéaire
— $f(n) = n.\log(n)$: complexité semi-logarithmique
— $f(n) = n^2$: complexité quadratique
— $f(n) = n^3$: complexité cubique
— $f(n) = n^p$: complexité polynomiale
— $f(n) = 2^n$: complexité exponentielle

Les fonctions usuelles correspondent à des algorithmes connus (les axes sont représentés avec une échelle logarithmique), Figure 1. Le Tableau présente les fonctions usuelles avec un exemple illustrant la complexité associée (Tableau 4).

2.4.2 Interprétation des données vis-à-vis du temps d'exécution

Temps d'exécution en fonction de la taille des données

Le temps d'exécution est quantifié pour les fonctions usuelles en fonction de la taille des données (Tableau 5).

Hypothèses : Nombre d'instructions par seconde = 10^6, ∞ pour valeur dépassant 10^{100}

Note : si $f(n) = 1$, temps d'exécution est de 1μs

Taille Maximale des données

Le tableau suivant (Tableau 6) propose d'évaluer la taille maximale des données en fonction du temps que l'on considère comme acceptble pour une application donnée.

Hypothèses : Nombre d'instructions par seconde = 10^6, ∞ pur valeur dépassant 10^{100}

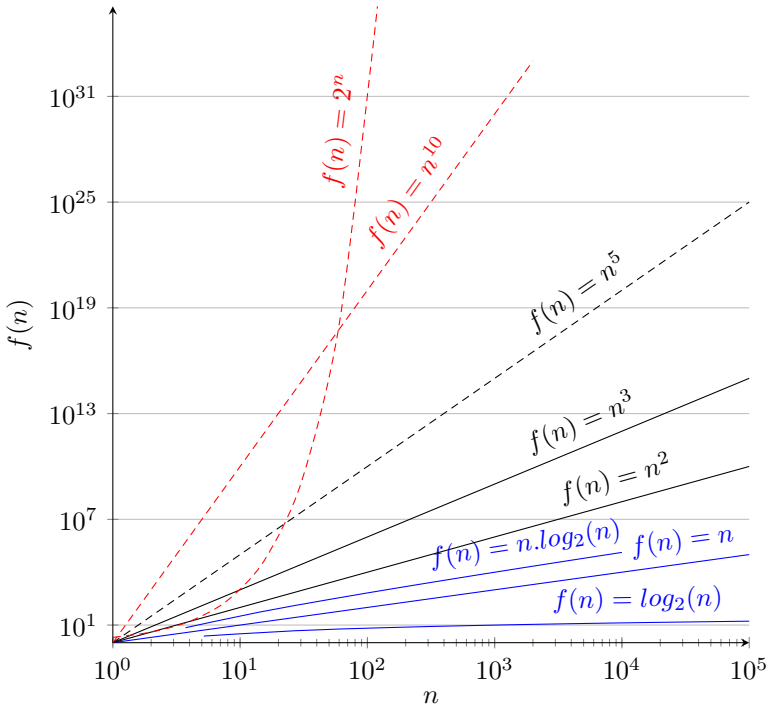


FIGURE 1 – Évolution des principales fonctions

2.4.3 Mesure de Performance et Illustration

Mesure de Performance

Exemple 8 : Évaluation d'un algorithme de tri (tri par insertion) sur deux machines différentes. Nous vous invitons à considérer l'annexe A.1.

— 486 DX 50 (processeur 80486 intel, 50 MHz) - 1990 (Windows) (figure 2)

— CPU quad-core 2.8 GHz Intel Core, 3 Go of RAM - 2014 (Windows) (figure 3)

Quelques remarques ...

— pour 10000 entiers : 40 min (80486 DX 50) à 0.2 seconde (quad core 2.8 GHz) (cas défavorable Tri par insertion)

— pour 500000 valeurs : 589 secondes (Tri par insertion) à 0.14 seconde (Tri Fusion) sur la même machine

ERREUR D'INTERPRÉTATION :
En mesure de performance, le choix des instances est primordial. Par exemple, supposons que les mesures (pour le tri par insertion dans le cas défavorable) sont effectuées jusqu'à 10000 valeurs entières, vous allez (peut-être ?) considérer

TABLE 4 – Illustration des principaux ordres de grandeur

Fonctions	Exemple d'Algorithme
$log(n)$	**Recherche Dichotomique** : En fonction de la valeur recherchée, sélectionner la partition gauche ou droite (après calcul d'une valeur pivot)
n	**Recherche séquentielle** : Rechercher une valeur en comparant élément par élément dans une liste
$n \cdot log_2(n)$	Complexité des meilleurs algorithmes de tri d'un tableau de n éléments. Par exemple, le **tri fusion** qui consiste à trier récursivement la moitié du tableau et les fusionner.
n^2	complexité dans le pire des cas pour le **Tri rapide** (*Quick-sort*). Cet algorithme se comporte en $n \cdot log_2(n)$ pour une instance déjà bien triée.
$n^{2.81}$	**Algorithme de Strassen** pour calculer le produit de deux matrices de taille $n \times n$
n^3	Algorithme naïf pour le **produit de deux matrices** de taille $n \times n$
n^k	complexité polynomiale lorsque le coût est borné par un polynôme (k ne dépend pas des entrées). On considère souvent que seuls les coûts bornés par un polynôme sont supportables en pratique.
2^n	**problème** SAT : meilleure complexité connue du test de satisfiabilité d'une formule propositionnelle à n variables.

TABLE 5 – Temps d'exécution des fonctions usuelles en fonction des données n

	Fonctions usuelles $f(n)$					
n	$log_2(n)$	n	$n.log_2(n)$	n^2	n^3	2^n
10^2	6.6μs	0.1 ms	0.6 ms	10 ms	1 sec	4.10^{16} a
10^3	9.9μs	1 ms	9.9 ms	1 sec	16.6 mn	∞
10^4	13.3μs	10 ms	0.1 sec	100 sec	11.5 j	∞
10^5	16.6μs	0.1 sec	1.6 sec	2.7 h	31.7 a	∞
10^6	19.9μs	1 sec	19.9 sec	11.5 j	31×10^3 a	∞

que la complexité pratique est de l'ordre du linéaire, et non du quadratique. Il faudrait augmenter très sensiblement le nombre de valeurs pour trouver « le bon résultat » tout en vérifiant que vous ne dépassez pas les limites de calcul de votre machine.

L'évolution technologique ne change pas la complexité théorique de l'algo-

TABLE 6 – Taille maximale des données pour un temps de calcul fixé

Temps Calcul	Fonctions usuelles					
	1, $log_2(n)$	n	$n.log_2(n)$	n^2	n^3	2^n
1 sec	∞	10^6	63×10^3	10^3	100	19
1 mn	∞	6×10^7	28×10^5	77×10^2	390	25
1 heure	∞	36×10^8	13×10^7	60×10^3	15×10^2	31
1 jour	∞	86×10^9	27×10^8	29×10^4	44×10^4	36

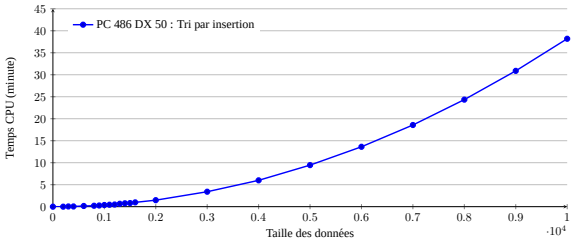


FIGURE 2 – Exemple de mesure de performance : 486 DX 50

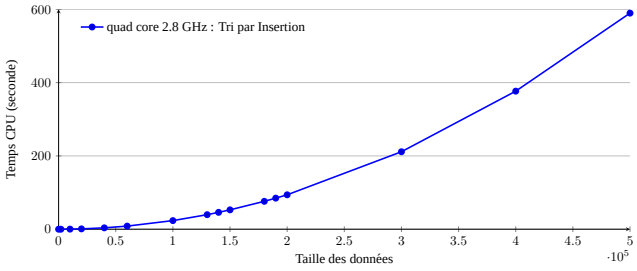


FIGURE 3 – Exemple de mesure de performance : quad core 2.8 GHz

rithme (le tri par insertion dans le cas défavorable reste $T(n) = a.n^2$).

Supposons les constantes a_i à l'itération i définies par $a_i = \frac{TM_i}{n_i^2}$. Nous pouvons déterminer la constante a par la moyenne[1] des a_i (nous obtenons dans notre exemple $a = 2.3476 \times 10^{-9}$) ; puis nous pouvons ainsi calculer les temps estimés TE_i à partir de cette constante (Tableau 7).

Il est possible d'estimer le temps que va prendre l'algorithme pour une instance quelconque

Exemple 9 : pour un tri par insertion $n = 1000000$, le temps estimé sera 2347 secondes (soit 39 minutes de temps CPU). Notre mesure du temps CPU est de 2369.89 secondes.

Exercice 3 : Reprendre l'algorithme du tri par insertion et effectuer les mesures de performances dans le cas favorable, défavorable et moyen. Appliquer la

1. La valeur associée à l'instance $n = 2000$ n'est pas prise en compte.

TABLE 7 – Détermination de la constante a et du temps estimé TE

n_i	Temps mesuré TM_i	constante a_i	Temps estimé TE_i
0	0	-	
2000	0.016	4×10^{-9}	0.0094
10000	0.234	2.34×10^{-9}	0.2348
20000	0.938	2.345×10^{-9}	0.939
40000	3.75	2.3438×10^{-9}	3.7562
60000	8.437	2.3436×10^{-9}	8.4514
100000	23.468	2.3468×10^{-9}	23.476
130000	24.468	2.3465×10^{-9}	39.675
140000	46.016	2.3478×10^{-9}	46.013
150000	52.828	2.3479×10^{-9}	52.821
180000	76.093	2.3485×10^{-9}	76.062
190000	84.797	2.3489×10^{-9}	84.749
200000	93.875	2.3469×10^{-9}	93.904
300000	211.36	2.3484×10^{-9}	211.28
400000	376.656	2.3541×10^{-9}	375.62
500000	589.563	2.358×10^{-9}	586.9

méthode précédente et comparer avec les résultats obtenus.

Exercice 4 :

1. On veut comparer les implémentations du tri par insertion et du tri par fusion sur la même machine. Pour un nombre n d'éléments à trier, le tri par insertion demande $8.n^2$, alors que le tri par fusion demande $64.n.\log_2(n)$. Quelles sont les valeurs de n pour lesquelles le tri par insertion l'emporte sur le tri par fusion
2. Quelle est la valeur minimale de n pour laquelle un algorithme dont le temps d'exécution est $100.n^2$, s'exécute plus vite qu'un algorithme dont le temps d'exécution est $2.n$, sur la même machine ?

Exercice 5 : Déterminer la complexité pour chaque définition de C_n^k. Le nombre de combinaisons C_n^k de k objets parmi n objets est défini par :

1. $C_n^k = \frac{n!}{k!.(n-k)!}$
2. $C_n^k = \frac{n.(n-1).\cdots.(n-k+1)}{k.(k-1).\cdots.1}$
3. Effectuer le calcul de C_n^k par C_n^{n-k} si $k > \frac{n}{2}$

2.5 Notations

Notations asymptotiques (Cf Annexe A.2 pour des compléments math.)

— Notion d'Ordre de grandeur

— Comparaison entre différents algorithmes
  — Algorithme 1 : Tri par insertion $\Theta(n^2)$
  — Algorithme 2 : Tri par fusion $\Theta(n.\log_2(n))$
  — Donc Algorithme 2 plus "rapide" que Algorithme 1 (n grand)

Objectif
ANALYSE ASYMPTOTIQUE DES ALGORITHMES

2.5.1 Notations usuelles : Θ, O, Ω

(i) Notation Θ (« Grand Theta »)

Definition 4.

$$\Theta\left(g\left(n\right)\right) = \{f(n) : \exists c_1, c_2, n_0 / 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n), \forall n \geq n_0\}$$

avec c_1, c_2, n_0 constantes positives

— Remarque(s)
  — $g(n)$ est une **borne asymptotiquement approchée** de $f(n)$
  — $f(n) \in \Theta(g(n))$: Par abus de langage : $f(n) = \Theta(g(n))$
  — $f(n)$ est dit **asymptotiquement positive** (*i.e.*, toujours positive pour n suffisamment grand)
  — $f(n)$, Θ : fonctions asymptotiquement positives

Illustrons la définition de Θ en figure 4.

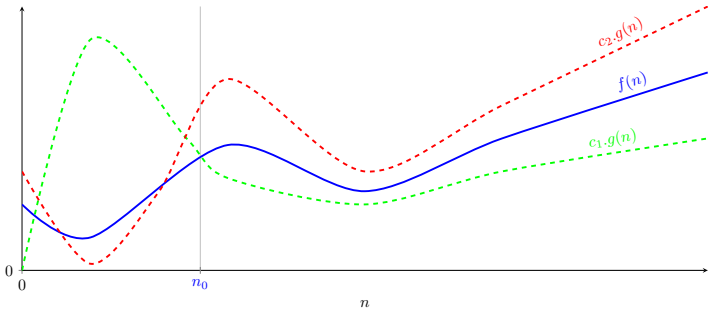


FIGURE 4 – Schéma explicatif de $\Theta\left(g\left(n\right)\right)$

Exemple 10 : Vérifier $\frac{1}{2}.n^2 - 3.n = \Theta(n^2)$?

Démonstration. Par définition, nous pouvons affirmer que :

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

$c_1 \leq \frac{1}{2} - \frac{3}{n}$ valide pour $n \geq 7, c_1 \geq \frac{1}{14}$
$\frac{1}{2} - \frac{3}{n} \leq c_2$ valide pour $n \geq 1, c_2 \geq \frac{1}{2}$

D'où : $n_0 = 7, c_1 = \frac{1}{14}, c_2 = \frac{1}{2}$

□

La figure 5 illustre l'exemple proposé et met en évidence l'encadrement.

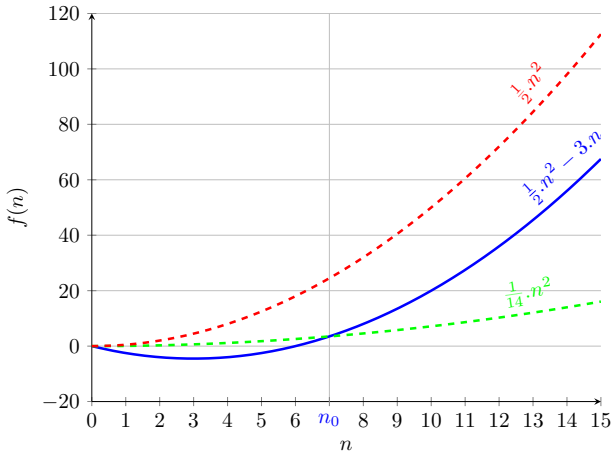


FIGURE 5 – Illustration pour $\frac{1}{2}.n^2 - 3.n$

Exemple 11 : Montrer que $6.n^3 = \Theta(n^2)$ n'est pas vrai pour n arbitrairement grand ?

Démonstration. Répondre à cette question revient à rechercher un entier n_0 vérifiant cette égalité, $6.n^3 = \Theta(n^2)$.

$6.n^3 = \Theta(n^2)$ signifie (par définition) que $6.n^3 \leq c_2 n^2$, d'où : $n \leq \frac{c_2}{6}$ avec $n \geq n_0$ et c_2 constant. Nous pouvons donc en conclure que ce n'est pas possible pour n arbitrairement grand.

□

Ignorer les termes d'ordre inférieur d'une fonction asymptotiquement positive pour le calcul des bornes asymptotiquement approchées

Exercice 6 : (à compléter) Soit $f(n) = a.n^2 + b.n + c, a > 0$ et a, b, c constantes ; Vérifier $f(n) = \Theta(n^2)$?

Démonstration. Pour une fonction quadratique, il est possible de montrer que :

- $c_1 = \frac{a}{4}$
- $c_2 = \frac{7.a}{4}$
- $n_0 = \max(\frac{|b|}{a}, \sqrt{\frac{|c|}{a}})$

□

Theorem 5. *Plus généralement : pour un polynôme* $p(n) = \sum_{i=0}^{d} a_i.n^i$, a_i *constantes et* $a_d > 0$, $p(n) = \Theta(n^d)$

Exemple 12 : $p(n) = a$ (a une constante) $p(n) = \Theta(n^0)$, *i.e.*, $p(n) = \Theta(1)$

Exercice 7 : Proposer une démonstration du théorème précédent en déterminant les différentes constantes.

Exercice 8 : Montrer que pour deux constantes réelles a et b quelconques $(b > 0)$, nous avons : $(n + a)^b = \theta\left(n^b\right)$.

(ii) Notation O (« Grand O »)

Definition 6.

$$O\left(g\left(n\right)\right) = \{f(n) : \exists c, n_0 / 0 \leq f(n) \leq c.g(n), \forall n \geq n_0\}$$

avec c,n_0 des constantes positives

- Remarque(s)
  - $(f(n) = \Theta(g(n))) \rightarrow (f(n) = O(g(n)))$

La figure 6 illustre la définition de O.

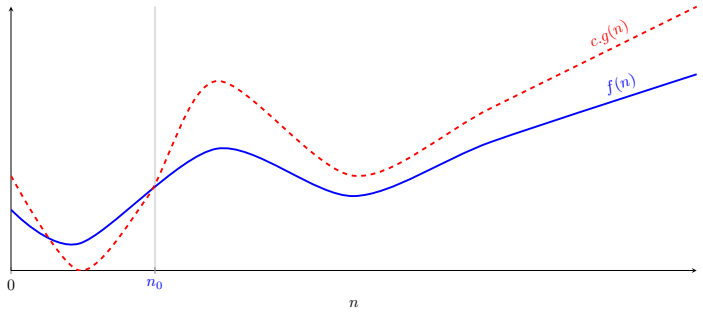


FIGURE 6 – Schéma explicatif de $O\left(g\left(n\right)\right)$

Exercice 9 : En reprenant la définition de $O\left(g\left(n\right)\right)$, les égalités suivantes $2^{n+1} = O(2^n)$? $2^{2.n} = O(2^n)$ sont-elles vérifiées ?

Démonstration.
- $2^{n+1} = O(2^n)$ est vérifiée si et seulement si $2^{n+1} \in O(2^n)$ est également vérifiée, c'est-à-dire $\exists c, n_0 / 0 \leq 2^{n+1} \leq c.2^n, \forall n \geq n_0$. Prenons $c = 2$ toujours vérifié pour $\forall n \geq n_0$.
- $2^{2.n} = O(2^n)$ est vérifiée ssi $\exists c, n_0 / 0 \leq 2^{2.n} \leq c.2^n, \forall n \geq n_0$. impossibilité de déterminer c_0 et n_0 pour l'encadrement.

□

Exercice 10 : Montrer que $17.x^2 + \sqrt{3.x - 4} = O(x^2)$.

Exemple 13 : Soit $f(n) = a.n^2 + b.n + c$, $a > 0$ avec a, b et c des constantes $(a > 0)$. Montrer les affirmations suivantes :

— $f(n) = \Theta(n^2)$
— $f(n) = O(n^2)$

Exercice 11 : Soit $f(n) = a.n + b$, $a > 0$ et a et b des constantes. Montrer que :
— $f(n) = \Theta(n)$
— $f(n) = O(n^2)$ avec $c = a + |b|$ et $n_0 = 1$
— ***AMUSANT !*** *Distinction entre « borne asymptotiquement approchée » et « borne supérieure asymptotique ».*

(iii) Notation Ω (« Grand Omega »)

Definition 7.

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 / 0 \leq c.g(n) \leq f(n), \forall n \geq n_0\}$$

avec c, n_0 constantes positives

Theorem 8.

$$f(n) = \Theta(g(n)) \text{ ssi } f(n) = O(g(n)) \text{ et } f(n) = \Omega(g(n))$$

Comme pour les autres définitions, la figure 7 illustre la définition de « Grand Omega ».

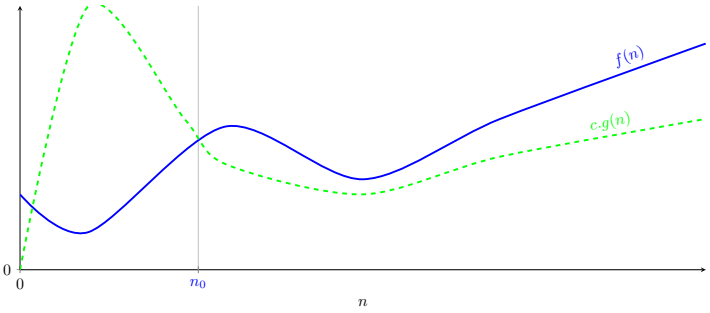


FIGURE 7 – Schéma explicatif de $\Omega(g(n))$

Exemple 14 : Soit $f(n) = a.n^2 + b.n + c, a > 0$ et a,b et c constantes
— $f(n) = \Theta(n^2)$
— Donc : $f(n) = O(n^2)$ et $f(n) = \Omega(n^2)$

Exercice 12 : Reprendre le tri par insertion et compléter l'analyse initiale (à faire), et en déduire :
— Cas optimal $f(n) = \Omega(n)$, et
— D'où : un temps d'exécution du tri par insertion compris entre $\Omega(n)$ et $O(n^2)$.

2.5.2 Autres Notations : o et ω

En général, il est plus difficile de déterminer les notations asymptotiques. Heureusement, les notations Θ, Ω et O sont souvent suffisants dans nos études.

(i) Notation o (« petit o »)

Definition 9.

$$o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 / 0 \leq f(n) < c.g(n), \forall n \geq n_0\}$$

Theorem 10. *f croît plus lentement que g :*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Exercice 13 : Vérifier les propriétés suivantes

- $2.n = o(n^2)$
- $2.n^2 \neq o(n^2)$
- $n! = o(n^n)$
- $17.x^2 + \sqrt{3.x - 4} = o(x^4)$

(ii) Notation ω (« petit omega »)

Definition 11.

$$\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 / 0 \leq c.g(n) < f(n), \forall n \geq n_0\}$$

Theorem 12.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$$

Exercice 14 : A Vérifier !

- $\frac{n^2}{2} = \omega(n)$
- $\frac{n^2}{2} \neq \omega(n^2)$
- $n! = \omega(2^n)$

2.5.3 Robustesse et Asymptotes

1. Interprétation des formules (en vue de simplifier)
   - $n = O(n^2) \overset{Signification}{\longrightarrow} n \in O(n^2)$
   - $2.n^2 + 3.n + 1 = 2.n^2 + \Theta(n) \overset{Signification}{\longrightarrow} 2.n^2 + 3.n + 1 = 2.n^2 + f(n)$
     - avec $f(n)$ fonction particulière de $\Theta(n)$
     - $f(n) = 3.n + 1$
2. $T(n) = 2.T(\frac{n}{2}) + \Theta(n)$?
   - Comportement asymptotique de $T(n)$
   - Fonction anonyme représentée par $\Theta(n)$
3. Interprétation de $2.n^2 + 3.n + 1 = 2.n^2 + \Theta(n) = \Theta(n^2)$?
   - $2.n^2 + 3.n + 1 = 2.n^2 + \Theta(n)$
     - il existe une certaine fonction : $f(n) \in \Theta(n)$
     - telle que $2.n^2 + 3.n + 1 = 2.n^2 + f(n)$
   - $2.n^2 + 3.n + 1 = \Theta(n^2)$
     - Pour toute fonction $(f(n))$: $g(n) \in \Theta(n)$
     - Il existe un certaine fonction : $h(n) \in \Theta(n^2)$
     - telle que $2.n^2 + g(n) = h(n)$

Robustesse des Notations Asymptotiques

Le tableau 8 résume les résultats obtenus. Par exemple, supposons que notre algorithme (appelé en l'occurrence A_1) a une complexité logarithmique. Nous avons estimé la taille maximale des données résolue par les ordinateurs à T_1. Nous pouvons en déduire, selon ce tableau, que la taille maximale des données serait T_1^{100} si notre ordinateur (dans un futur proche) pourrait être 100 fois pus rapide. Un raisonnement identique est valable pour les autres classes de fonctions usuelles représentées par les algorithmes (de A_2 à A_6).

TABLE 8 – Robustesse des notations

Algor.	Complexité	Taille max. résolue par les machines actuelles (notée T_i)	Taille max. résolue en 100 fois plus rapide (notée Z_i)
A_1	$\log(n)$	T_1	$Z_1 = T_1^{100}$
A_2	n	T_2	$Z_2 = 100.T_2$
A_3	$n.\log(n)$	T_3	$Z_3 = 100.T_3$
A_4	n^2	T_4	$Z_4 == 10.T_4$
A_5	2^n	T_5	$Z_5 = T_5 + \log(100)$
A_6	$n!$	T_6	$Z_6 = T_6 + \frac{\log(100)}{\log(T_6)-1}$

La relation entre Z_i et T_i est définie par $100.f(T_i) = f(Z_i)$, où $f(.)$ est une fonction asymptotique.

Exemple 15 : Pour l'algorithme A_1, nous obtenons $100.\log(T_1) = \log(Z_1)$, i.e., $\log_2(T_1^{100}) = \log_2(Z_1)$; d'où $Z_1 = T_1^{100}$.

Exercice 15 : Montrer le résultat pour l'algorithme A_6.

Démonstration. Nous obtenons $100.T_6! = Z_6!$. Pour les grandes valeurs de n, nous pouvons appliquer la formule de Stirling :

$$n! = \left(\frac{n}{e}\right)^n$$

Par conséquent, la relation devient :

$$100.\left(\frac{T_6}{e}\right)^{T_6} = \left(\frac{Z_6}{e}\right)^{Z_6}$$

En introduisant la fonction $\log_2$, on obtient :

$$\log(100) + T_6.(\log(T_6) - 1) = Z_6(\log(Z_6))$$

En posant $Z_6 = T_6 + \epsilon$, et en approximant $\log(T_6 + \epsilon)$ par $\log(T_6)$, pour des petites valeurs de ϵ, nous obtenons :

$$Z_6 = T_6 + \frac{\log(100)}{\log(T_6) - 1}$$

□

Exercice 16 : Vérifier les autres relations du Tableau 8.

2.6 Pour aller plus loin ...

Pour aller plus loin dans la compréhension des chapitres, nous donnerons quelques exercices supplémentaires :

Exercice 17 : Proposer un algorithme qui donne une solution au problème $TWO + TWO = FOUR$. A chaque lettre est associée une valeur entre 0 et 9. En déterminer sa complexité.

Exercice 18 : Nous souhaitons déterminer la complexité temps/mémoire de trois algorithmes différents permettant de calculer la somme suivante (ne proposez pas d'algorithme récursif) :

$$\sum_{k=1}^{n} \frac{x^k}{k!}, x \neq 0$$

Exercice 19 : Proposer un algorithme pour le produit de deux matrices ainsi que pour l'algorithme de Strassen. En déduire leur complexité respective.

Exercice 20 : Déterminer un algorithme qui recherche les m entiers (les valeurs de m sont comprises entre 1 et 9) dont la somme vaut S. En déduire la complexité de l'algorithme proposé.

Exercice 21 : Un nombre parfait est un nombre qui est égal à la somme de ses diviseurs différents de lui-même. Le nombre 6 est un nombre parfait ; car il est divisible par $1, 2, 3$ et $1 + 2 + 3 = 6$.

1. Déterminer un algorithme recherchant tous les nombres parfaits. Calculer la complexité temporelle de cet algorithme
2. Une amélioration possible est de considérer que les nombres pairs. Il semble en effet que le nombre parfait impair n'existe pas (actuellement, aucune preuve mathématique le confirme). Quelle est la conséquence de cette amélioration sur la complexité ?

2.7 Conclusion

Conclusion

— Approche intuitive de la complexité (temps, mémoire).

— Notion d'ordre de grandeur (notation asymptotique).

— Privilégier l'étude du cas défavorable (cas favorable, défavorable et moyen).

— A FAIRE : revoir la complexité pour le tri par insertion.

— A COMPLÉTER : voir l'analyse probabiliste pour les cas moyens, et essayer de mesurer la performance pour le tri par insertion.

3 Algorithme de type « Diviser pour Régner »

3.1 Introduction

Prenons un exemple illustrant les algorithme « Diviser pour Régner », le tri fusion.

Un exemple du Tri par fusion

Procédure Fusion(A,p,q,r)

```
-.1  début
-.2      //n1 : longueur de A[p..q], n2 :longueur de A[q+1..r] ;
-.3      n1 ← q − p + 1 ; n2 ← r − q ;
-.4      Creer_Tableaux L[1..n1+1] et R[1..n2+1] ;
-.5      pour i ← 1 à n1 faire
-.6          L[i] ← A[p + i − 1] ;
-.7      pour j ← 1 à n2 faire
-.8          R[j] ← A[q + j];
-.9      L[n1 + 1] ← ∞ ; R[n2 + 1] ← ∞ i ← 1 ; j ← 1 ;
-.10     pour k ← p à r faire
-.11         si L[i] ≤ R[j] alors
-.12             A[k] ← L[i] ; i = i + 1 ;
-.13         sinon
-.14             A[k] ← R[j] ; j = j + 1 ;
```

Exercice 22 : Montrer que la complexité de la procédure *Fusion* est en $\Theta(n)$

Exercice 23 : Améliorer l'algorithme en supprimant les instructions de contrôle des *sentinelles* (lignes définies par $L[n1 + 1] \leftarrow \infty$ et $R[n2 + 1] \leftarrow \infty$). Vérifier que la complexité de la procédure *Fusion* ne change pas.

Tri par fusion

Algorithme 2 : Tri_Fusion(A,p,r)

```
     Entrées : A[1..n] non triée , p, r : entier
     Sorties : A[1..n] triée
2.1  début
2.2      si p < r alors
2.3          q ← ⌊(p+r)/2⌋ ;
2.4          Tri_fusion (A,p,q) ;
2.5          Tri_fusion (A,q + 1, r) ;
2.6          Fusion (A,p,q,r)
```

— Question : Complexité temporelle du Tri_fusion ?

— Réponse : Algorithme de la famille *Diviser pour Régner*

3.2 Caractéristiques des Algorithmes *Diviser pour Régner*

Identifier un algorithme *Diviser pour Régner*

1. Diviser le problème en a sous-problèmes de taille $\frac{n}{b}$
2. Régner sur les sous-problèmes en les résolvant récursivement
3. Combiner la solution des sous-problèmes pour produire la solution du problème

Definition 13. Un algorithme « Diviser pour Régner » est défini par une relation de récurrence de la forme :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq c, \\ a.T(\frac{n}{b}) + D(n) + C(n) & \text{sinon} \end{cases} \quad (1)$$

Équation de Récurrence de l'Algorithme Tri Fusion

— $T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ 2.T(\frac{n}{2}) + \Theta(n) & \text{sinon} \end{cases}$

— Ré-écriture : c Temps Requis pour un problème de taille 1

— $T(n) = \begin{cases} c & \text{si } n = 1, \\ 2.T(\frac{n}{2}) + c.n & \text{sinon} \end{cases}$

Objectif
Présentation de différentes méthodes pour déterminer la complexité d'un algorithme décrit par une relation de récurrence.

Simplification de la Notation

Exemple 16 : Supposons un algorithme décrit par la relation de récurrence :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(n) & \text{sinon} \end{cases}$$

— $T(n)$ défini pour n entier

— si n est suffisamment petit, $T(n)$ est constant

Simplification par :

$$T(n) = 2.T(\frac{n}{2}) + \Theta(n)$$

3.3 Substitution

3.3.1 Principe de la méthode par substitution

Méthode par Substitution : Principe

Principe en 2 phases

1. Conjecturer la forme de la solution
2. Employer une récurrence mathématique pour trouver les constantes et prouver que la solution est correcte

Méthode puissante (borner par récurrence par excès ou par défaut), mais ne peut s'utiliser que lorsque la réponse est facile à deviner.

Exemple 17 : Supposons que nous souhaitons déterminer la complexité de l'algorithme décrit par la relation suivante :

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

Démonstration. Il est nécessaire pour la méthode par substitution de proposer et de vérifier la conjecture envisagée.

1. Phase 1 : Conjecturer la forme de la solution
   - Proche de l'algo. Tri Fusion $T(n) = 2.T(\frac{n}{2}) + \Theta(n)$.
   - Donc une Conjecture envisagée pour la complexité est $\Theta(n.\log_2(n))$.
2. Phase 2 : Vérification
   - Prouver $T(n) \leq c.n.\log_2(n)$, $c > 0$?
   - Supposons vraie $T(\lfloor \frac{n}{2} \rfloor) \leq c.\frac{n}{2}.\log_2(\lfloor \frac{n}{2} \rfloor)$
   - $T(n) \leq 2.\left(c.\lfloor \frac{n}{2} \rfloor.\log_2(\lfloor \frac{n}{2} \rfloor)\right) + n$
   - $T(n) \leq c.n.\log_2(\lfloor \frac{n}{2} \rfloor) + n$
   - $T(n) \leq c.n.\log_2(n) - c.n.\log_2(2) + n$
   - $T(n) \leq c.n.\log_2(n) - c.n + n$
   - $T(n) \leq c.n.\log_2(n)$, si $c \geq 1$ cqfd.

□

Solution valable aux conditions aux limites ?

Après vérification de la conjecture, il faut vérifier les solutions aux limites considérées comme des cas initiaux.

Démonstration. L'objectif consiste ici à déterminer les différentes constantes (c et n_0).

- Choix de c pour vérifier $T(n) \leq c.n.\log_2(n)$,$c \geq 1$
- $T(1) = 2.T(\lfloor \frac{1}{2} \rfloor) + 1 = 1$

— $T(1) \leq c.1.\log_2(1) = 0$

— contradiction

Astuce choix de n_0 :
Distinction entre le cas initial de la récurrence ($n = 1$) et le cas de la démonstration ($n = 2$, $n = 3$)

Prenons $n_0 = 2$

1. $T(2) = 2.T(\lfloor \frac{2}{2} \rfloor) + 2 = 4$, donc $T(2) \leq c.2.\log_2(2)$
2. $T(3) = 2.T(\lfloor \frac{3}{2} \rfloor) + 3 = 5$, donc $T(3) \leq c.3.\log_2(3)$

Choix de c pour vérifier les cas initiaux : $c \geq 2$

□

Comment Conjecturer les solutions de récurrence

Pas de règles générales pour effectuer une conjecture.

(i) Intuition ! ! !

— *Exercice 24 :* $T(n) = 2.T(\lfloor \frac{n}{2} \rfloor + 17) + n$?

— conseil : proche de $T(n) = 2.T(\lfloor \frac{n}{2} \rfloor) + n$,
donc conjecture $T(n) = \Theta(n.\log_2(n))$

(ii) Réduire les Bornes (supérieure et inférieure)

— *Exercice 25 :* $T(n) = 2.T(\lfloor \frac{n}{2} \rfloor) + n$?

— Borne Inf. : $T(n) = \Omega(n)$ et Borne Sup. : $T(n) = O(n^2)$. On peut ensuite réduire l'intervalle pour converger vers la conjecture $T(n) = \Theta(n.\log_2(n))$

3.3.2 Problèmes classiques rencontrés pour la méthode par substitution

Du mauvais choix d'une conjecture

Exemple 18 : Supposons la relation de récurrence décrit par $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1$

Démonstration. Dans un premier temps, nous pourrions proposer une conjecture en complexité linéaire (qui va s'avérer fausse).

— Phase 1 : Conjecture proposée $T(n) = O(n)$

— Phase 2 : Prouver que $T(n) \leq c.n$

Essayons de prouver $T(n) \leq c.n$?

— $T(n) \leq c.\lfloor \frac{n}{2} \rfloor + c.\lceil \frac{n}{2} \rceil + 1$

— $T(n) \leq c.n + 1$, échec de la conjecture

Proposition : Nous pourrions essayer une conjecture d'ordre supérieur $T(n) = O(n^2)$, qui se révélera être aussi une « grossière » ERREUR.

En fait, il existe une Conjecture bonne à la constante prés. L'idée repose sur l'introduction d'un terme constant inférieur à notre hypothèse initiale. Il faudrait prouver que que $T(n) \leq c.n-b, b \geq 0$, i.e. $T(n) \leq c.\lfloor \frac{n}{2} \rfloor - b + c.\lceil \frac{n}{2} \rceil - b + 1 = c.n - 2.b + 1$. Nous prouvons ainsi que $T(n) \leq c.n - b$. □

Du changement de variables

Exemple 19 : Cet exemple illustre le changement de variables. Soit la relation définie par $T(n) = 2.T(\sqrt{n}) + \log_2(n)$. Déterminer sa conjecture.

Démonstration. Supposons le changement de variable $m = \log_2(n)$, $T(m) = 2.T(2^{\frac{m}{2}}) + m$. Posons $S(m) = T(2^m)$, $S(m) = S(\frac{m}{2}) + m$. Cette relation est connue (tri fusion), et nous pouvons admettre comme conjecture : $O(m.\log_2(m))$. En repassant de $S(m)$ à $T(n)$, nous en déduisons : $T(n) = T(2^m) = S(m) = O(m.\log_2(m)) = O(\log_2(n).\log_2(\log_2(n)))$

□

3.4 Arbres recursifs

3.4.1 Principe de la Méthode des arbres récursifs

Méthode par Arbre Récursif

Principe en 3 phases
Méthode de construction d'arbre permettant de trouver une « bonne conjecture ».

1. Déterminer le coût de chaque nœud (coût d'un sous-problème individuel)
2. Totaliser les coûts pour chaque niveau de l'arbre
3. Cumuler les coûts de chaque niveau pour obtenir le coût total, puis vérifier avec la méthode de substitution

3.4.2 Un Exemple simple : le Tri Fusion

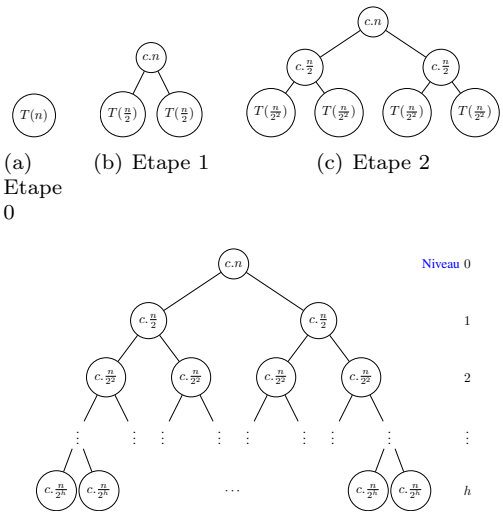
Application à l'Algorithme du Tri Fusion

Prenons un exemple simple (l'algorithme du tri fusion) pour expliquer la méthode des arbres récursifs.

Exemple 20 : $T(n) = \begin{cases} c & \text{si } n = 1, \\ 2.T(\frac{n}{2}) + c.n & \text{sinon} \end{cases}$

Phase 1 : coût de chaque nœud L'objectif consiste à déterminer le coût de chaque nœud par application de la règle de récurrence.

Démonstration. Par application de la relation de récurrence, nous obtenons les principales étapes pour la construction de l'arbre récursif du Tri Fusion (Figure 8). Sa construction permet de déterminer le coût de chaque nœud.



(a) Etape 0 (b) Etape 1 (c) Etape 2

(d) Dernière Etape

FIGURE 8 – Phase 1 : Construction de l'Arbre Récursif pour le Tri Fusion

□

Phase 2 : coût de chaque niveau Après avoir déterminé le coût de chaque noeud par application de la relation de récurrence, il faut calculer le coût de chaque niveau : en recherchant la hauteur, les niveaux et le coût de ces derniers (sans oublier le coût des feuilles).

Démonstration. Nous donnons les différents éléments pour évaluer le coût de chaque niveau (figure 9 [2])

1. Hauteur de l'Arbre : $n \longrightarrow \frac{n}{2} \longrightarrow \frac{n}{4} \longrightarrow \ldots \longrightarrow 1$

$$\frac{n}{2^h} = 1 \textbf{ d'où } h = \log_2(n) \textbf{ (avec } h \textbf{ : hauteur)}$$

2. Nombre de Niveaux et coûts : $\log_2(n) + 1$ $(0, 1, 2, \ldots, \log_2(n))$, à chaque niveau, le coût est de $c.n$

2. arbre en Tikz repris de Manuel Kirsch. Un grand Merci !

3. Coûts du niveau terminal : Rappelons que chaque niveau a 2^i nœuds. Au niveau de la feuille (i.e. de profondeur $\log_2(n)$) a un nombre de nœuds estimé à

$$2^h = 2^{\log_2(n)} = n$$

□

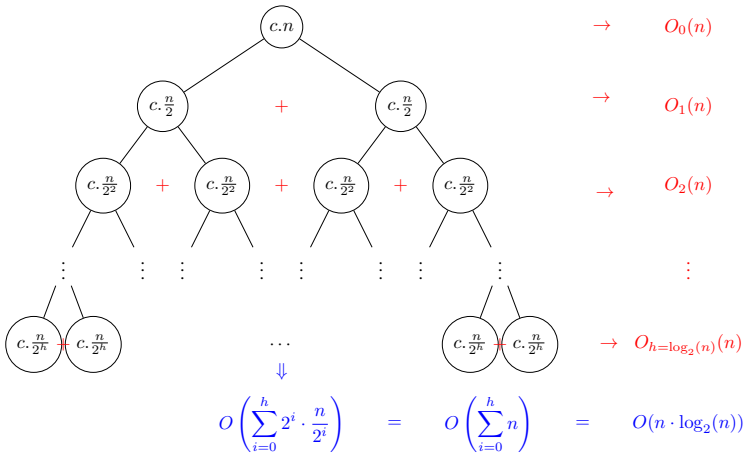


FIGURE 9 – Schéma résultant du tri fusion

Phase 3 : coût total , $T(n) = \sum_{i=0}^{h-1} cout_i + cout_h$ (avec $cout_i$: coût au niveau i, $cout_h$: coût des feuilles)

Démonstration. Nous pouvons évaluer maintenant le coût total.

$$T(n) = c.n + c.n + \ldots + \Theta(n)$$
$$T(n) = c.n + c.n + \ldots + \Theta(n)$$
$$T(n) = \sum_{i=0}^{\log_2(n)-1}(c.n) + \Theta(n)$$
$$T(n) = c.n.\sum_{i=0}^{\log_2(n)} 1$$

Donc $$T(n) = \Theta(n.\log_2(n))$$

□

En résumé, nous obtenons :

Hauteur	$\log_2(n)$
Niveau	$\log_2(n)+1$
coût d'un niveau i	$c.n$
coût du niveau racine	$c.n$
coût total	$T(n) = c.n.(\log_2(n)+1)$
Conjecture	$\Theta(n.\log_2(n))$

Application de la méthode de substitution pour vérifier la validité de la conjecture : déjà fait.

3.4.3 Un Deuxième exemple où $a \neq b$

2eme Exemple : $T(n) = 3.T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$

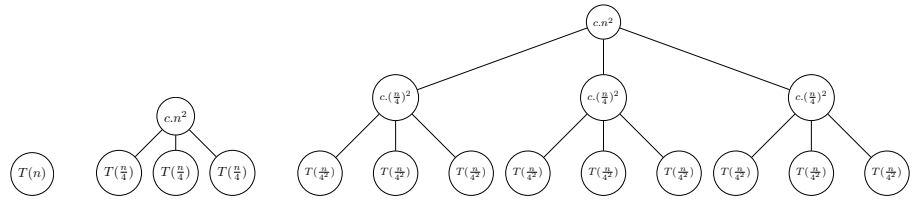
Exemple 21 : $T(n) = 3.T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$

Pour rappel, la partie entière n'est pas prise en compte pour la résolution des récurrences :

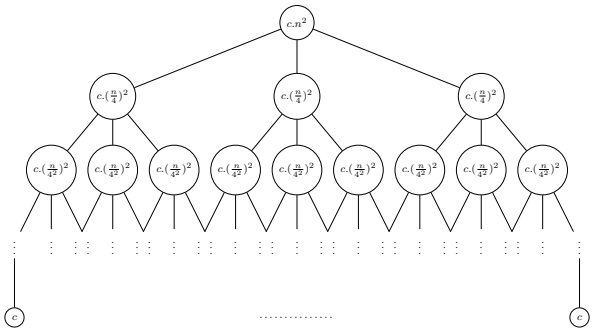
$$T(n) = 3.T(\frac{n}{4}) + c.n^2$$

Phase 1 : coût de chaque nœud Par application de la relation de récurrence, nous construisons l'arbre récursif associé à cette relation de récurrence.

Démonstration. Nous obtenons les principales étapes pour la construction de l'arbre récursif (Figure 10). Sa construction permet de déterminer le coût de chaque nœud.



(a) Etape 0 (b) Etape 1 (c) Etape 2



(d) Dernière Etape

FIGURE 10 – Phase 1 : Construction de l'Arbre Récursif pour le 2nd exemple

□

Phase 2 : coût de chaque niveau De manière similaire, nous pouvons évaluer le coût de chaque niveau.

Démonstration. Calculons le coût de chaque niveau (Figure 11) en recherchant la hauteur, les niveaux et le coût de ces derniers (sans oublier le coût des feuilles).

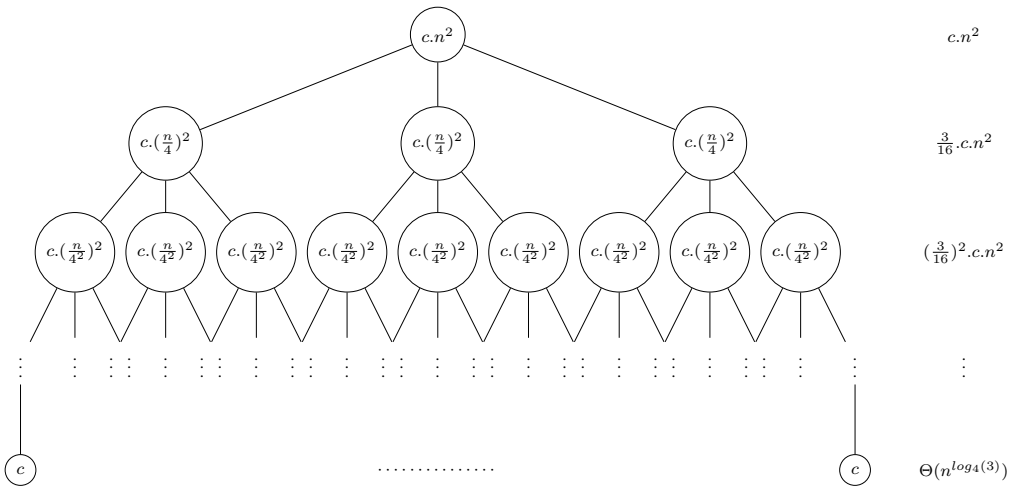


FIGURE 11 – Arbre récursif pour $T(n) = 3.T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$

1. Hauteur de l'Arbre : $n \longrightarrow \frac{n}{4} \longrightarrow \frac{n}{16} \longrightarrow \ldots \longrightarrow 1$

$$\frac{n}{4^h} = 1 \textbf{ d'où } h = \log_4(n) \textbf{ (avec } h \textbf{ : hauteur)}$$

2. Coûts des niveaux : $\log_4(n) + 1$ (i.e. $0, 1, 2, \ldots, \log_4(n)$). Chaque niveau i a un nombre de 3^i nœuds
3. Coûts du niveau terminal : Au niveau de la feuille (racine), i.e. de profondeur $\log_4(n)$ a un nombre de nœuds :

$$3^{\log_4(n)} = n^{\log_4(3)} \textbf{, puisque } a^{\log_b(c)} = c^{\log_b(a)}$$

.

□

Phase 3 : coût total Nous pouvons déterminer son coût total.

Démonstration. Le coût total est déterminé par :

$$T(n) = c.n^2 + \frac{3}{16}.c.n^2 + (\frac{3}{16})^2.c.n^2 + \ldots + \Theta(n^{\log_4(3)})$$
$$T(n) = \sum_{i=0}^{\log_4(n)-1}(\frac{3}{16})^i.c.n^2 + \Theta(n^{\log_4(3)})$$

$$T(n) = \frac{(\frac{3}{16})^{\log_4(n)}-1}{\frac{3}{16}-1}c.n^2 + \Theta(n^{\log_4(3)})$$

□

Or $\sum_{i=0}^{\log_4(n)-1}(\frac{3}{16})^i.c.n^2 + \Theta(n^{\log_4(3)}) < \sum_{i=0}^{\infty}(\frac{3}{16})^i.c.n^2 + \Theta(n^{\log_4(3)})$,
avec $T(n) = \sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ pour $|x| < 1$

Donc $T(n) < \frac{16}{13}.c.n^2 + \Theta(n^{\log_4(3)})$

En résumé, nous obtenons :

Hauteur	$\log_4(n)$
Niveau	$\log_4(n)+1$
coût d'un niveau i	$(\frac{3}{16})^i.c.n^2$
coût du niveau racine	$\Theta(n^{\log_4(3)})$
coût total	$T(n) < \frac{16}{13}.c.n^2 + \Theta(n^{\log_4(3)})$
Conjecture	$T(n) = O(n^2)$

Appliquer la méthode de substitution pour vérifier la validité de la conjecture

Démonstration. Prouver que $T(n) \leq d.n^2$, $d > 0$?
$T(n) \leq 3.T(\lfloor \frac{n}{4} \rfloor) + c.n^2 \leq 3.d.(\lfloor \frac{n}{4} \rfloor)^2 + c.n^2$
$T(n) \leq (\frac{n}{4})^2 + c.n^2 = (\frac{3}{16}).d.n^2 + c.n^2$
D'où : $T(n) \leq d.n^2$ Vérifiée si $d \geq \frac{16}{13}.c$ □

3.4.4 3ème Exemple : Revenons sur l'algorithme tri fusion

Exercice 26 : Supposons que nous proposons un tri fusion avec deux partitions $\frac{1}{3}$ pour la première partition (la seconde est un partitionnement en $\frac{2}{3}$). Définir la relation de récurrence, construire l'arbre récursif et en déterminer sa complexité. à faire. Pour vous aider dans votre démarche, nous vous donnons quelques conseils :

- relation de récurrence pour la nouvelle version du tri fusion : $T(n) = T(\frac{n}{3}) + T(\frac{2.n}{3}) + O(n)$ (arbre récursif proposé en Figure 12)
- Hauteur : prendre le plus long chemin $n \longrightarrow \frac{2}{3}.n \longrightarrow (\frac{2}{3})^2.n \longrightarrow \ldots \longrightarrow 1$ (Nous avons évalué le cout des feuilles à la figure 12)

$$(\frac{2}{3})^h.n = 1 \textbf{ d'où } h = \log_{\frac{3}{2}}(n)$$

.

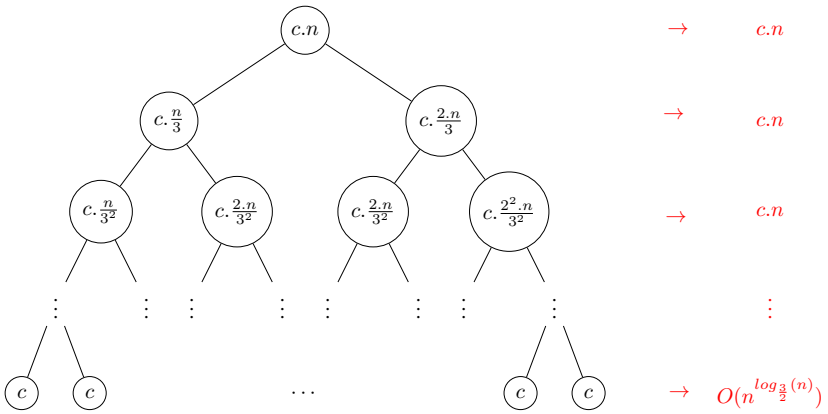


FIGURE 12 – Arbre récursif

— Prendre l'hypothèse que le coût est uniformément distribué pour les différents niveaux de l'arbre (En fait, l'arbre n'est pas un arbre binaire complet et ne donne pas toujours $c.n$). Déterminer sa complexité.

— Effectuer un travail similaire pour déterminer sa complexité en prenant le plus court chemin pour calculer h.

— EN déduire la Conjecture : $O(n.\log_2(n))$

3.5 Methode Générale

3.5.1 Principe de la méthode

Méthode Générale (Bentley, 1980**) [14]**

Principe de la Méthode Générale
Recette pour résoudre les récurrences de la forme
$T(n) = a.T(\frac{n}{b}) + f(n)$, $a \geq 1$ et $b > 1$
Idée de la méthode générale : Comparaison des fonctions (asymptotiquement) entre $f(n)$ et $n^{\log_b(a)}$

1. $n^{\log_b(a)} > f(n)$ (cas 1 du théorème)
2. $n^{\log_b(a)} = f(n)$ (cas 2 du théorème)
3. $n^{\log_b(a)} < f(n)$ (cas 3 du théorème)

Exemple 22 : : Tri fusion
$a = 2$, $b = 2$, $f(n) = C(n) + D(n) = \Theta(n)$

Theorem 14. *Soient les constantes $a \geq 1$ et $b > 1$ et une fonction $f(n)$. Soit $T(n)$ défini pour les entiers non négatifs par la récurrence :*

$$T(n) = a.T(\frac{n}{b}) + f(n) \tag{2}$$

où l'on interprète $\frac{n}{b}$ *comme signifiant* $\lfloor \frac{n}{b} \rfloor$ *ou* $\lceil \frac{n}{b} \rceil$.

$T(n)$ *peut alors être borné asymptotiquement de la façon suivante :*

1. *Si* $f(n) = O(n^{\log_b(a)-\epsilon})$ *pour une constante* $\epsilon > 0$ *alors* $T(n) = \Theta(n^{\log_b(a)})$
2. *Si* $f(n) = \Theta(n^{\log_b(a)})$ *alors* $T(n) = \Theta(n^{\log_b(a)}.\log_2(n))$
3. *Si* $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ *pour une constante* $\epsilon > 0$ *et* $a.f(\frac{n}{b}) \leq c.f(n)$ *pour une constante* $c < 1$ *et* n *grand alors* $T(n) = \Theta(f(n))$

3.5.2 Illustration de la méthode générale

Application de la Méthode Générale

Exemple 23 : $T(n) = 9.T(\frac{n}{3}) + n$

Démonstration. $a = 9$, $b = 3$, $f(n) = n$
$n^{\log_b(a)} = n^{\log_3(9)} = n^{2.\log_3(3)} = n^2$
Application cas 1 :
$f(n) = n^{\log_b(a)-\epsilon}$, avec $\epsilon = 1$
Donc : $T(n) = \Theta(n^2)$

□

Exemple 24 : $T(n) = T(\frac{2.n}{3}) + 1$

Démonstration. $a = 1$, $b = \frac{3}{2}$, $f(n) = 1$
$n^{\log_b(a)} = n^{\log_{\frac{3}{2}}(1)} = n^0 = 1$
$f(n) = \Theta(n^{\log_b(a)})$
Application du cas 2 :
Donc : $T(n) = \Theta(n^{\log_{\frac{3}{2}}(1)}.\log_2(n)) = \Theta(\log_2(n))$ □

Exemple 25 : $T(n) = 16.T(\frac{n}{4}) + n^3$

Démonstration. $a = 16$, $b = 4$, $f(n) = n^3$
$n^{\log_b(a)} = n^{\log_4(16)} = n^2$

— $f(n) = \Omega(n^{\log_4(16)+\epsilon})$, avec $\epsilon = 1$
— $a.f(\frac{n}{b}) \leq c.f(n)$, $16.(\frac{n}{4})^3 \leq c.n^3$, d'où $\frac{1}{4} \leq c \leq 1$

Application du cas 3 :
Donc : $T(n) = \Theta(f(n)) = \Theta(n^3)$ □

Exemple 26 : et Quid de l'algorithme Tri Fusion ?

Démonstration. $a = 2$, $b = 2$, $f(n) = \Theta(n)$
$n^{\log_b(a)} = n^{\log_2(2)} = n$
Application cas 1 : échec
$f(n) = O(n^{1-\epsilon})$, avec $\epsilon > 0$ échec
Application cas 2 : $f(n) = \Theta(n)$, Réussite
Donc : Nous avons montré que le tri fusion a une complexité en $T(n) = \Theta(n^{\log_b(a)}.\log_2(n)) = n.\log_2(n)$. □

Exemple 27 : Tentative Application de la Méthode Générale sur la relation suivante : $T(n) = 2.T(\frac{n}{2}) + n.\log_2(n)$

Démonstration. $a = 2$, $b = 2$, $f(n) = n.\log_2(n)$
$n^{\log_b(a)} = n^{\log_2(2)} = n$
Application cas 1 : échec
Application cas 2 : échec
Application cas 3 :
$f(n)$ asymptotiquement plus grande que n, mais elle n'est pas polynomialement plus grande ($\frac{f(n)}{n^{\log_b(a)}} = \frac{n.\log_2(n)}{n} = \log_2(n)$) qui est asymptotiquement plus petite que n^{ϵ}, avec $\epsilon > 0$. échec
D'où **impossibilité d'appliquer la méthode générale**. □

3.6 Autres methodes

Autres Méthodes

Theorem 15. *si* $f(n) = \Theta(n^{\log_b(a)}.\log_2^k(n))$, $k \geq 0$
alors $T(n) = \Theta(n^{\log_b(a)}.\log_2^{k+1}(n))$

Exercice 27 :

— Montrer ce résultat (pour simplifier - analyse aux puissances exactes de k)

— Application de ce théorème sur l'exemple précédent $T(n) = 2.T(\frac{n}{2}) + n.\log_2(n)$

Theorem 16. *Théorème proposé par Akra et Bazzi (1998) qui permet de résoudre des récurrences (avec q divisions du problème en des sous-problèmes de tailles très inégales)*
$T(n) = \sum_{i=1}^{k} a_i.T(\lfloor \frac{n}{b_i} \rfloor) + f(n)$, *avec* $k \geq 1$, $a_i > 0$, $\sum a_i \geq 1$ *et* $b_i > 1$
$f(n)$ *est bornée, positive et non décroissante*
$\forall c > 0, \exists n_0, d > 0 / f(\frac{n}{c}) \geq d.f(n), \forall n \geq n_0$

1. *Déterminer la valeur de p (unique et positive) telle que* $\sum_{i=1}^{k}(a_i.b_i^{-p}) = 1$
2. *la solution de la récurrence est alors* $T(n) = \Theta(n^p) + \Theta(n^p.\int_{n'}^{n} \frac{f(x)}{x^{p+1}}dx)$ *avec* n' *suffisamment grand*

Exercice 28 : $T(n) = T(\lfloor \frac{n}{3} \rfloor) + T(\lfloor \frac{2.n}{3} \rfloor) + \Theta(n)$

— Montrer que la méthode générale n'est pas applicable

— Appliquer le théorème de Akra et Bazzi

3.7 Pour aller plus loin ...

Pour aller plus loin dans la compréhension des chapitres, nous donnerons quelques exercices supplémentaires :

Exercice 29 : N'hésitez pas à lire les papiers de Bentley [14] et de Akra et Bazzi [13].

Exercice 30 : Donner des bornes asymptotiques inférieure et supérieure pour $T(n)$. On supposera que $T(n)$ est constant pour $n < 3$.

1. $T(n) = 2.T(\frac{n}{2}) + n^3$
2. $T(n) = 4.T(\frac{9.n}{10}) + n$
3. $T(n) = 16.T(\frac{n}{4}) + n^2$
4. $T(n) = 7.T(\frac{n}{3}) + n^2$
5. $T(n) = 2.T(\frac{n}{4}) + \sqrt{n}$
6. $T(n) = T(n-1) + 4$
7. $T(n) = T(\sqrt{n}) + 1$
8. $T(n) = 3.T(\frac{n}{2}) + n.\log_2(n)$
9. $T(n) = 5.T(\frac{n}{5}) + \frac{n}{\log_2(n)}$
10. $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + T(\frac{n}{8})$
11. $T(n) = T(n-1) + \frac{1}{n}$

Exercice 31 : Puissances

— Écrivez un programme permettant de calculer x^n à partir de la définition $x^n = x * x^{n-1}$

— Déterminez la complexité théorique de votre algorithme et tracez la courbe correspondante.

— Modifiez votre algorithme de manière à utiliser la formule de récurrence suivante :

$$x^n = \begin{cases} x^{\frac{n}{2}} \times x^{\frac{n}{2}} & si \quad n \quad pair \\ x \times x^{\frac{n}{2}} \times x^{\frac{n}{2}} & si \quad n \quad impair \end{cases}$$

— Quelle est la complexité de votre nouvel algorithme ?

Exercice 32 : D'après le théorème de Fermat (démontré par J. Wilkes en 1999), pour x, y, z entiers naturels, l'équation $x^n + y^n = z^n, n > 2$ n'a pas de solution.

1. Proposer un algorithme de recherche exhaustive des valeurs entières x, y, z, n. Calculer la complexité temporelle de cet algorithme trivial. Le calcul de la puissance d'un nombre par un nombre entier est aussi un algorithme trivial.
2. Nous souhaitons en premier abord améliorer le calcul de la puissance. Proposer un algorithme qui permet d'évaluer ce calcul suivant la valeur de n. Si n est impair, $x^n = x \times x^{n-1}$; Si n est pair, $x^n = (x^{\frac{n}{2}})^2$. Déterminer alors sa complexité ainsi que celle de l'algorithme global.
3. Au lieu d'effectuer une recherche exhaustive, il est possible d'utiliser les propriétés de commutativité de l'addition (x et y) et d'arrêter la recherche globale pour des valeurs de $y < x$. Il est tout à fait possible de rechercher un nombre z dont sa valeur est supérieure aux valeurs x et y. Quelle est dans ce cas la complexité temporelle de l'algorithme modifié.
4. La résolution de ce problème a conduit à des recherches mathématiques suivant la valeur de n. Lorsque n prend la valeur 2, l'équation prend 2 variables à gauche de l'égalité. Plus généralement, pour $n = i$, le nombre de variables x_i est i à gauche de l'égalité. Effectuer un travail similaire pour une équation du type $x_1^i + x_2^i + \ldots + x_i^i = z^i$.

Exercice 33 : Le **problème du cavalier** consiste à parcourir avec un cavalier l'ensemble des cases de l'échiquier, à partir d'une case de l'échiquier (i, j). Ce problème est de même nature que le problème des n-reines (la profondeur de l'arbre de récursion est cette fois-ci en n^2 au lieu de n pour les n-reines). Dans ces conditions, nous avons besoin d'une heuristique efficace : Elle s'appuie sur le nombre de cases $A(i, j)$ accessibles par le cavalier quand il est en position (i, j). L'heuristique consiste à déplacer d'abord le cavalier vers les cases dont le nombre d'accès est minimal, tout en décrémentant les autres cases accessibles de la case initiale.

1. Décrire l'algorithme de déplacement du cavalier
2. En déduire sa complexité

Exercice 34 : Problème du « Carré Magique » (extrait de [4]). Supposons un carré divisé en cases égales contenant un nombre entier. Ce carré est dit "magique" si la somme des nombres suivant chaque ligne, chaque colonne et chaque diagonale est toujours la même. Un exemple est illustré ci-dessous : Ce carré comporte 9 cases (il est dit "de carré 3") et la somme d'une ligne, d'une diagonale ou d'une colonne est de 15 :

2	9	4
7	5	3
6	1	8

1. Proposer un algorithme trivial générant un carré magique de N. En déterminer sa complexité.

2. Une méthode particulière permet de construire des carrés magiques d'ordre impair. Soit à construire un carré magique de 5. A l'extérieur d'un carré de 25 cases, formons des cases disposées en échelons et égales à celles du carré. Remplissons les 5 échelons de 5 cases ainsi formées avec les 25 premiers nombres dans leur ordre naturel.

.	.	.	.	1	.	.	.	.
.	.	.	6	.	2	.	.	.
.	.	**11**	.	**7**	.	**3**	.	.
.	16	.	**12**	.	**8**	.	4	.
21	.	**17**	.	**13**	.	**9**	.	5
.	22	.	**18**	.	**14**	.	10	.
.	.	**23**	.	**19**	.	**15**	.	.
.	.	.	24	.	20	.	.	.
.	.	.	.	25	.	.	.	.

Puis laissant intacts les nombres inscrits de cette façon à l'intérieur du carré (en gras), reportons ceux de l'extérieur dans la colonne où ils se trouvent déjà et dans la cinquième case après celle où on les a écrits".

11	24	7	20	3
4	12	25	8	16
17	5	13	21	9
10	18	1	14	22
23	6	19	2	15

Déterminer la complexité de l'algorithme proposé par cette méthode pour N quelconque impair.

3. Rectangles magiques : Un rectangle est dit "magique" si les éléments de chacune de ses lignes ont une même somme, et si les éléments de chacune de ses colonnes ont aussi une somme constante, qui sera en général différente de la première. Par exemple un rectangle de 5×3 est décrit ci-dessous (somme à 40 pour les lignes, 24 pour les colonnes) :

1	13	10	4	12
15	9	3	6	7
8	2	11	14	5

Exercice 35 : Un arbitre dépose au hasard (i.e., *au hasard* signifie que l'arbitre choisit une couleur de manière équitable) des chapeaux sur les têtes de n joueurs. Un joueur ne peut voir le chapeau qu'il a sur la tête. Les couleurs des chapeaux sont prises parmi n, mais plusieurs des chapeaux utilisés peuvent avoir la même couleur. L'arbitre interroge alors les n joueurs de l'assemblée qui répondent en même temps. Les joueurs ont pu avant le jeu convenir d'une méthode de jeu, mais pendant le jeu, ils n'échangent aucune information. Chaque joueur essaie de deviner la couleur du chapeau qu'il a sur la tête.

Proposer un algorithme qui permet aux joueurs de gagner avec un gain de 100%. Calculer la complexité de l'algorithme.

Exercice 36 : Soit n le nombre de nœuds d'un arbre binaire. Trouver la relation de récurrence pour déterminer le nombre d'arbres binaires possibles pour un n donné et calculer la complexité de l'algorithme.

3.8 Conclusion

Conclusion

— A RETENIR

1. Obtention de bornes asymptotiques à partir de récurrences pour des algorithmes de type « Diviser pour Régner »
2. Description de plusieurs méthodes (substitution, arbre récursif, méthode générale)
3. Illustration sur plusieurs exemples (dont le tri par fusion)

— A FAIRE : différents exercices non traitées

— A COMPLÉTER : voir la preuve du théorème de la méthode générale

4 NP-complétude

4.1 Introduction

Repositionner notre problématique en 2 catégories

- Problèmes « **faciles** » résolus en temps *Polynomial* sont (Nous pouvons affirmer qu'il existe une borne sup. en n^k).
  - Existence d'algorithmes efficaces (avec k petit)
  - **P** signifie **Déterministe Polynomial**.
- Problèmes « difficiles » nécessitant un temps **supra-polynomial** (besoin d'un temps de calcul important même pour des *petites entrées*).
  - Absence d'algorithmes efficaces
  - souvent possèdent une structure combinatoire
  - **NP** signifie **Non déterministe Polynomial**.

Exemple 28 :

- Fonctions **polynomiales** : $\log_2(n)$, n, n^2, n^3, $n^k + c_1.n^{k-1} + \cdots + c_{k-1}$
- Fonctions **supra-polynomiales** (au mieux des fonctions exponentielles) : $1.001^n + n^6$, 5^n, n^n, $n!$, $n^{\log_2(n)}$

4.2 Définitions

classes P et NP

Definition 17 (classe **P**)**.** La classe **P** (Polynomial) définit l'ensemble des problèmes pouvant être résolu de manière **déterministe** en temps polynomial.

Exemple 29 : Pour démontrer qu'un problème appartient à la classe **P**, la manière la plus simple est d'exhiber un algorithme polynomial qui résout ce problème.

Mais, il est plus difficile de prouver qu'un problème n'appartient pas à **P**, car ne pas trouver d'algorithme ne veut pas dire qu'il n'en existe pas. Il a fallu attendre 2002 pour prouver que le problème de la recherche des nombres premiers est dans la classe **P**, algorithme **AKS** (découvert par Pr. Agrawal et al.) [12].

Definition 18 (classe **NP**)**.** La classe **NP** (Non-déterministe Polynomial) définit l'ensemble des problèmes pouvant être résolu de manière **non-déterministe** en temps polynomial.

Exemple 30 : Pratiquement, il existe pour ces problèmes un algorithme déterministe polynomial qui permet de vérifier une solution au problème. Pour les instances positives, il existe un « certificat d'appartenance » de taille polynomiale qui peut être vérifié en un temps polynomial.

Definition 19 (classe **NP – complet**)**.** La classe **NP – complet** (Non-déterministe Polynomial) définit l'ensemble des problèmes de la classe **NP** qui vérifie que pour deux problèmes P_1 et P_2 pouvant être résolus de manière **non-déterministe** en temps polynomial.

4.3 Etude de cas : la tournée du Facteur

Application : « Une Histoire de Facteur »

La **Tournée du facteur** (**TF**) consiste à aller de maison en maison pour distribuer du courrier.

- Apporter de Bonnes Nouvelles
- Et dans un temps *raisonnable* en minimisant (si possible) les distances (Les distances entre les maisons sont non nulles).

Son stage d'évaluation aura lieu dans une ville très bourgeoise *PlaisantVille*

Exemple 31 : Pour son concours, il doit distribuer le courrier des habitants de la rue principale dont la configuration est particulière (Figure 13).

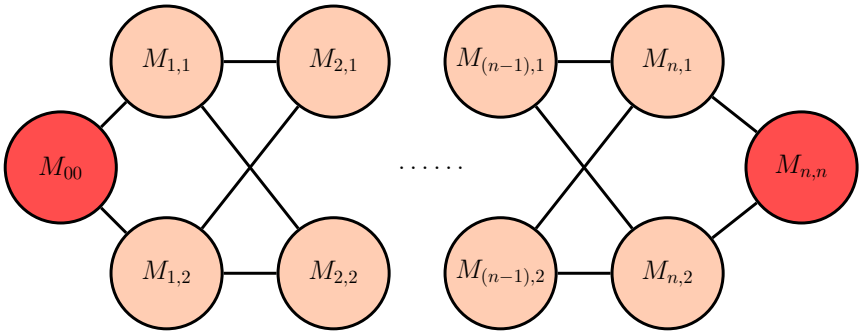


FIGURE 13 – Schéma de la rue principale

M_{00} : la poste ; M_{ij} une maison, $M_{n,n}$: la dernière maison

QUESTION : Pouvez vous aider ce facteur à prendre de bonne décision en terme de minimisation des distances ?

4.3.1 Itérer tous les chemins possibles

Démonstration. Une première idée consiste à itérer tous les chemins possibles pour pouvoir évaluer les différents chemins possibles, $nb(M_{0,0}, M_{n,n})$:

- $nb(M_{1,1}, M_{n,n}) = nb(M_{1,2}, M_{n,n})$, donc $nb(M_{0,0}, M_{n,n}) = 2.nb(M_{1,1}, M_{n,n})$
- $nb(M_{2,1}, M_{n,n}) = nb(M_{2,2}, M_{n,n})$, donc $nb(M_{0,0}, M_{n,n}) = 2.2.nb(M_{2,1}, M_{n,n})$
- etc.
- d'où : $nb(M_{0,0}, M_{n,n}) = 2^n$

Conclusion : Algo. Itérer sur tous les chemins possibles est en $\Theta(2^n)$. □

4.3.2 2eme approche : Recherche du plus court chemin de manière incrémentale

Démonstration. Une deuxième idée vise à prendre à chaque cycle le chemin le plus court. Par convention, nous allons admettre qu'il existe une certaine distance entre deux maisons reliées directement par un chemin, appelé c :

$$mini(M_{n,1}, M_{n,n}) = c(M_{n,1}, M_{n,n}), \text{ et } mini(N_{n,2}, M_{n,n}) = c(N_{n,2}, M_{n,n}) \quad (3)$$

Algorithme 3 : plus_court_chemin (G) : réel

Entrées : G graphe valué

3.1 **début**

3.2 **pour** $i \leftarrow n-1$ *à* 1 **faire**

3.3 $mini(M_{i,1}, M_{n,n}) \leftarrow \min(mini(M_{(i+1),1}, M_{n,n}) + c(M_{i,1}, M_{(i+1),1}), mini(M_{(i+1),1}, M_{n,n}) + c(M_{i,2}, M_{(i+1),2}))$;

3.4 $mini(M_{i,2}, M_{n,n}) \leftarrow \min(mini(M_{0,1}, M_{i,1}) + c(M_{i,1}, M_{(i+1),1}), mini(M_{0,1}, M_{i,1}) + c(M_{i,1}, M_{(i+1),1}))$;

3.5 $plus\_court\_chemin \leftarrow \min(c(M_{0,0}, M_{i,1}) + mini(M_{i,1}, M_{n,n}), c(M_{0,0}, M_{i,2}) + mini(M_{i,2}, M_{n,n}))$

Conclusion : Il existe un Algo pour Rechercher le min de tous les chemins possibles est $\Theta(n)$ en temps (**Il existe bien un algorithme polynomial**).

□

Décision du facteur

Il choisit le 2nd Algorithme qui permet de minimiser les distances en $\Theta(n)$. **Le facteur a trouvé un algorithme polynomial** et réussit ainsi son concours.

Graphe Valué quelconque

Exemple 32 : Gestion d'une Ville - Ayant réussi son concours d'évaluation (il a trouvé un temps polynomial pour sa tournée dans une rue principale), le responsable lui propose de gérer le courrier de tous les citoyens M_i d'une petite ville comme *PlaisantVille*, i.e. un graphe valué (Figure **??**).

Question (problème plus difficile pour notre facteur) :

Trouver un circuit (le plus court) passant par toutes les maisons pour la distribution du courrier et revenant à sa base (la poste) ?

4.3.3 Itérer tous les cycles possibles sur un graphe quelconque

Une première idée consiste à itérer sur tous les cycles.

Démonstration. Un algorithme naïf est estimé en Figure 14.

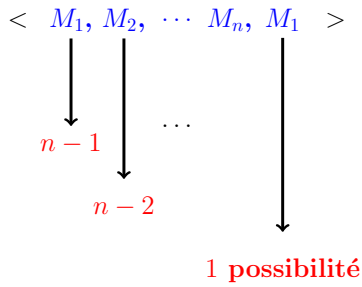


FIGURE 14 – Recherche exhaustive

Conclusion : Algorithme trivial donne $T(n) = (n-1)!$. □

Peut-on faire mieux ? Pas d'Algo. plus performant en Polynomial.

4.3.4 Vérifier qu'un circuit est inférieur à une contrainte de distance

Il est possible de **vérifier** qu'un circuit est inférieur à une certaine distance.

Démonstration. Le facteur propose de vérifier qu'un chemin particulier est inférieur à une distance donnée (facile à montrer qu'il existe un algorithme en temps polynomial) :

$$\langle M_1, M_2, \ldots, M_n, M_1 \rangle / distance(\langle M_1, M_2, \ldots, M_n, M_1 \rangle) \leq a$$

Conclusion : Algo existant en Polynomial. □

4.3.5 Trouver un circuit inférieur à une contrainte de distance

Trouver un circuit inférieur à une certaine distance

Démonstration. Le facteur propose de trouver un chemin dont la distance serait inférieure à une certaine constante :

$$\langle M_1, M_2, \ldots, M_n, M_1 \rangle / distance(\langle M_1, M_2, \ldots, M_n, M_1 \rangle) \leq a$$

Conclusion : Itération sur tous les cycles et vérifier la contrainte sur la constante $T(n) = \Theta(n!)$ □

Peut-on faire mieux ? Pas d'Algo plus performant en Polynomial

Avis du Facteur sur l'étude de cas
Il n'existe pas d'Algo plus performant que l'algo trivial précédent. Son problème est donc un problème de Décision

4.4 Pb Décision

4.4.1 Notion

Décider ...

Théorie de la **NP**-complétude correspond à un problème de décision :

Definition 20. Un problème de décision (noté π) consiste en (Figure 15)

— Un ensemble des instances D_π

— Un sous ensemble des réponses positives $Y_\pi \subseteq D_\pi$.

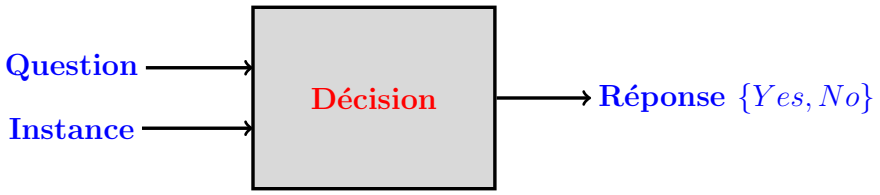


FIGURE 15 – Entrées/Sorties d'un problème de Décision

Revenons au problème du facteur

Exemple 33 : Le facteur a l'idée de formaliser son problème comme un problème de décision (noté π) se définissant ainsi :

— Instance

  — ensemble de maisons $\{M_1, M_2, \ldots, M_n\}$

  — distance $d(M_i, M_j) \in \aleph$

  — constante a

— Question

  — existe t-il une tournée ?

$$\langle M_{\pi_1}, M_{\pi_2}, \ldots, M_{\pi_n}, M_{\pi_1} \rangle / \sum_{i=1}^{n-1} (d(M_{\pi_i}, M_{\pi_{i+1}})) + d(M_{\pi_n}, M_{\pi_1}) \leq a$$

Definition 21. Rappel : La décomposition classique peut se définir par un triplet $\langle A, F, L \rangle$ (Figure 16).

Correspondance entre Décision et Langage
En pratique : Exécuter l'instance sur quoi ? comment ?

$$L(\pi, e) = \{x \in F / x \text{ encodage par } e \text{ d'une instance } I \in Y_\pi\}$$

— Instance du problème $\overset{Encodage}{\longrightarrow}$ Problème de décision

— $D_\pi \longrightarrow L$

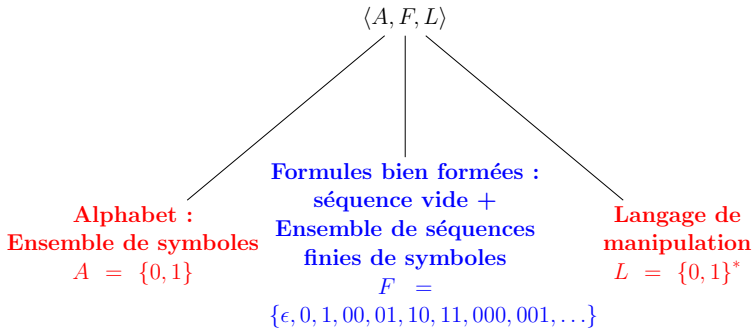


FIGURE 16 – Décomposition classique du triplet

— $Y_\pi \longrightarrow L(\pi, e) \subseteq L$

Encodage e

Exemple 34 : Graphe orienté (Figure 17)
:

— Instance : Graphe orienté pondéré par les arêtes

— Question : . . . ?

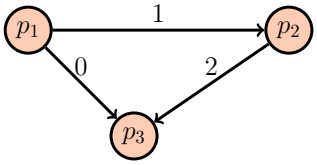


FIGURE 17 – Graphe valué simple à 3 nœuds/ 3 arêtes

encodage <arêtes, poids> :
$((([01],[10]),([10],[11]),([01],[11])),([01],[10],[00]))$
Indépendant de l'encodage

— *Exemple 35 :* Pour un tableau de n éléments : Utilisation d'une fonction polynomiale $longueur : D_\pi \rightarrow \aleph$ indépendante de l'encodage e

Pour information : pour le problème du facteur $longueur(I) = m + \lceil \log_2(a) \rceil + \max\{\lceil \log_2(d(c_i, c_j)) \rceil, \forall c_i, c_j\}$

4.4.2 Machine de Turing

Machine de Turing Déterministe (DTM)

Définition formelle Une machine M de Turing est formellement décrite par un t-uple (pour une discussion plus détaillée, voir par exemple [11]) :

$$M = (Q, \Gamma, \Sigma, \delta, s, B, F) \tag{4}$$

où :

— Q est un ensemble fini d'états

— Γ est l'alphabet du ruban (alphabet utilisé sur le ruban)

— $\Sigma \subseteq \Gamma$ est l'alphabet d'entrée (l'alphabet utilisé pour le mot d'entrée)

— $s \in Q$ est l'état initial

— $B \in \Gamma - \Sigma$ est le symbole blanc (souvent dénoté # ou b)

— $F \subseteq Q$ est l'ensemble des états accepteurs

— $\delta : Q \times F \to Q \times \Gamma \times \{L, R\}$ est la fonction de transition (L et R sont utilisés pour représenter respectivement un déplacement de la tête de lecture vers la gauche (*left*) ou vers la droite (*right*)).

Concepts de Langage et de dérivation Comme pour les autres classes d'automates (Figure 18), le langage accepté par une machine de Turing se définit à l'aide des notions de configuration et de dérivation entre configuration. Une configuration contient toute l'information nécessaire pour poursuivre l'exécution, i.e. : (i) l'état de la machine, (ii) le mot apparaissant sur le ruban avant la position de la tête de lecture, et (iii) le mot se trouvant sur le ruban entre la position de la tête de lecture et le dernier caractère non blanc. Plus formellement, une configuration est un élément de :

$$Q \times \Gamma^* \times (\epsilon \cup \Gamma^*) \tag{5}$$

avec $\Gamma^* = \Gamma - \{B\}$, ϵ la tête de lecture est au début du ruban et une transition vers la gauche n'est pas possible (si le ruban n'est pas infini à gauche). Un exemple de ruban infini à gauche et à droite est proposé en Figure 19 [3].

Les configurations correspondantes sont respectivement :

— (q, α_1, α_2) avec (i) l'état de la machine q, (ii) α_1 est le mot apparaissant sur le ruban avant la position de la tête de lecture, et (iii) α_2 est le mot se trouvant sur le ruban entre la position de la tête de lecture et le dernier caractère non blanc.

— (q, α_1, ϵ) avec la différence sur le point (iii) : ϵ est la séquence vide se trouvant sur le ruban entre la position de la tête de lecture et le dernier caractère non blanc.

Soit la configuration (q, α_1, α_2). Écrivons cette configuration sous la forme $(q, \alpha_1, b\alpha_2')$ en prenant $b = \#$ dans le cas où $\alpha_2 = \epsilon$. Le concept de dérivation en une étape peut alors se définir pour cette configuration, par :

— Si $\delta(q, b) = (q', b', R)$, nous avons $(q, \alpha_1, b\alpha_2') \vdash_M (q', \alpha_1 b', \alpha_2')$

3. Merci à Sebastian Sardina pour le schéma d'une machine de Turing en TikZ.

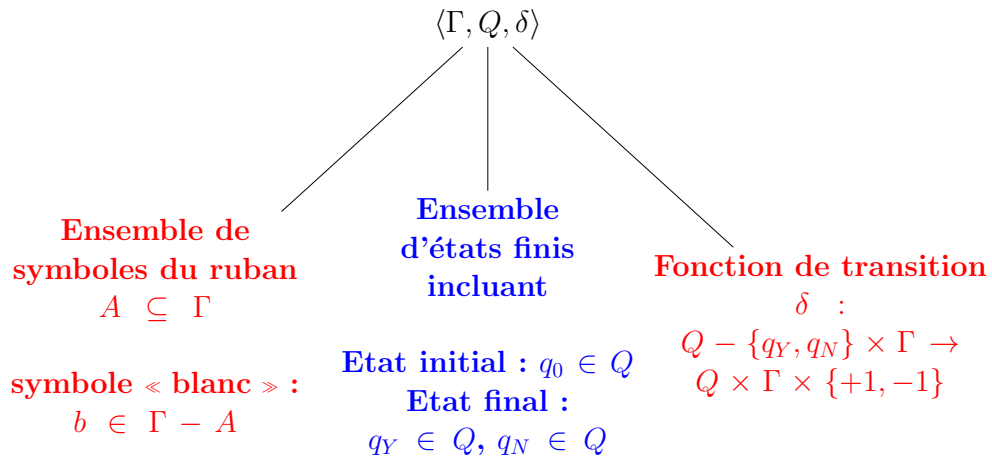


FIGURE 18 – Adaptation pour la DTM

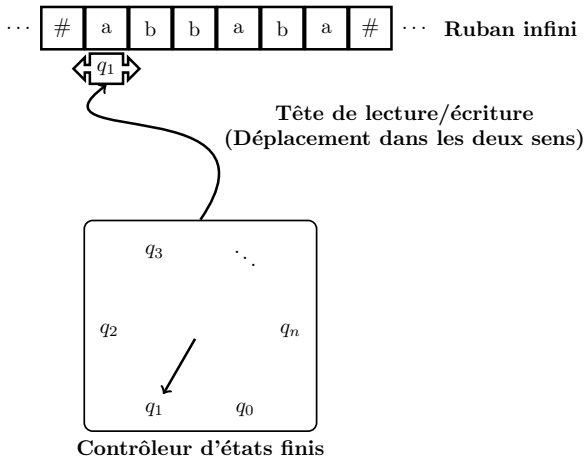


FIGURE 19 – Une Machine de Turing Déterministe avec ruban infini à gauche et à droite

— Si $\delta(q, b) = (q', b', L)$ et si $\alpha_1 \neq \epsilon$, et nous avons $(q, \alpha'_1 a, b\alpha'_2) \vdash_M (q', \alpha'_1, ab'\alpha'_2)$

La notion de dérivation en plusieurs étapes pourrait se définir pour la machine M pour la configuration initiale C par $C \vdash_M C'$ s'il existe $k \geq 0$ et des configurations intermédiaires $C_0, C_1, C_2, \cdots, C_k$ telle que :

— $C = C_0$

— $C' = C_k$

— $C_i \vdash_M C_{i+1}$ pour $0 \leq i < k$

Principe d'une DTM est décrit dans l'algorithme 4.

Algorithme 4 : DTM

Entrées : $x \in F$ Symbole placé sur la case 1, les autres cases à b ;
q Tête de lecture case 1

début
  $q \leftarrow q_0$, lire état s (case 1);
  tant que *état* $q \neq \{q_Y, q_N\}$ **faire**
    $q \in Q - \{q_Y, q_N\}, \delta(q', s', \Delta)$ est défini;
    effacer s;
    écrire s';
    déplacer case à gauche ($\Delta = -1$) ou à droite ($\Delta = 1$);
    passage à l'état q';
    $q \leftarrow q'$;
    $s \leftarrow s'$;
  $q = q_Y : Yes$; $q = q_N : No$;

Deux Exemples illustrant une DTM

Un premier exemple de la Machine de Turing Déterministe (DTM)

Exemple 36 :

— $\Gamma = \{0, 1, b\}$, $A = \{0, 1\}$, $Q = \{q_0, q_1, q_2, q_3, q_Y, q_N\}$

— Supposons $x = 10100$ comme instance d'entrée (Figure 20)

$\delta(q, s)$	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_N, b, -1)$
q_2	$(q_Y, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$
q_3	$(q_N, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$

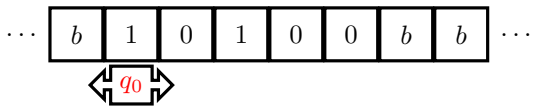


FIGURE 20 – État initial de la DTM

La suite détaille le déroulement de l'algorithme DTM en différentes étapes :

étape 1 **(Figure 21)**

$\delta(q, s)$	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_N, b, -1)$
q_2	$(q_Y, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$
q_3	$(q_N, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$

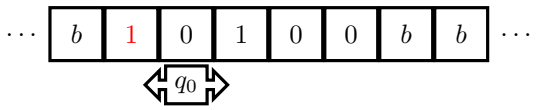


FIGURE 21 – étape 1 de la DTM

étape 2 **(Figure 22)**

$\delta(q,s)$	0	1	b
q_0	$(q_0,0,+1)$	$(q_0,1,+1)$	$(q_1,b,-1)$
q_1	$(q_2,b,-1)$	$(q_3,b,-1)$	$(q_N,b,-1)$
q_2	$(q_Y,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$
q_3	$(q_N,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$

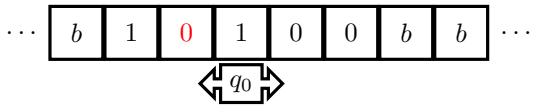


FIGURE 22 – étape 2 de la DTM

étape 3 **(Figure 23)**

$\delta(q,s)$	0	1	b
q_0	$(q_0,0,+1)$	$(q_0,1,+1)$	$(q_1,b,-1)$
q_1	$(q_2,b,-1)$	$(q_3,b,-1)$	$(q_N,b,-1)$
q_2	$(q_Y,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$
q_3	$(q_N,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$

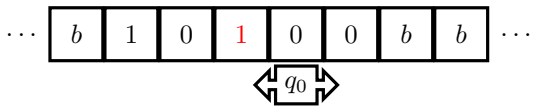


FIGURE 23 – étape 3 de la DTM

étape 4 **(Figure 24)**

$\delta(q,s)$	0	1	b
q_0	$(q_0,0,+1)$	$(q_0,1,+1)$	$(q_1,b,-1)$
q_1	$(q_2,b,-1)$	$(q_3,b,-1)$	$(q_N,b,-1)$
q_2	$(q_Y,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$
q_3	$(q_N,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$

étape 5 **(Figure 25)**

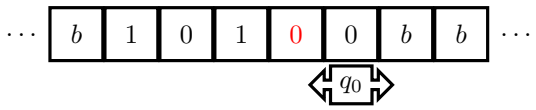


FIGURE 24 – étape 4 de la DTM

$\delta(q,s)$	0	1	b
q_0	$(q_0,0,+1)$	$(q_0,1,+1)$	$(q_1,b,-1)$
q_1	$(q_2,b,-1)$	$(q_3,b,-1)$	$(q_N,b,-1)$
q_2	$(q_Y,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$
q_3	$(q_N,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$

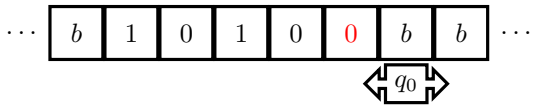


FIGURE 25 – étape 5 de la DTM

étape 6 **(Figure 26)**

$\delta(q,s)$	0	1	b
q_0	$(q_0,0,+1)$	$(q_0,1,+1)$	$(q_1,b,-1)$
q_1	$(q_2,b,-1)$	$(q_3,b,-1)$	$(q_N,b,-1)$
q_2	$(q_Y,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$
q_3	$(q_N,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$

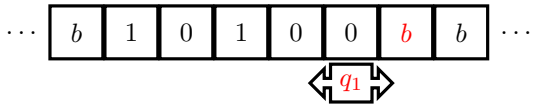


FIGURE 26 – étape 6 de la DTM

étape 7 **(Figure 27)**

$\delta(q,s)$	0	1	b
q_0	$(q_0,0,+1)$	$(q_0,1,+1)$	$(q_1,b,-1)$
q_1	$(q_2,b,-1)$	$(q_3,b,-1)$	$(q_N,b,-1)$
q_2	$(q_Y,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$
q_3	$(q_N,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$

étape 8 **(Figure 28)**

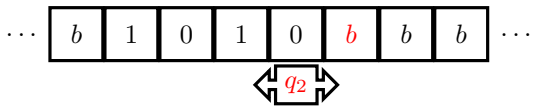


FIGURE 27 – étape 7 de la DTM

$\delta(q,s)$	0	1	b
q_0	$(q_0,0,+1)$	$(q_0,1,+1)$	$(q_1,b,-1)$
q_1	$(q_2,b,-1)$	$(q_3,b,-1)$	$(q_N,b,-1)$
q_2	$(q_Y,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$
q_3	$(q_N,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$

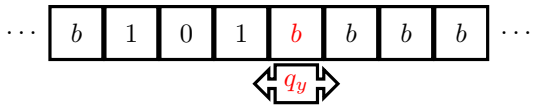


FIGURE 28 – étape 8 de la DTM

Conclusion du DTM

Réponse pour $x = 10100$ **est** Yes **en** 8 **étapes**

Exercice 37 : effectuer un travail similaire avec $x = 00110$

Un deuxième exemple de la Machine de Turing Déterministe (DTM)

Exercice 38 : Un exemple plus général, mais le mécanisme est similaire aux exemples précédents. Soit la machine de Turing $M = (Q, \Gamma, \Sigma, \delta, s, B, F)$ où :

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- $\Gamma = \{a, b, X, Y, \#\}$
- $\Sigma = \{a, b\}$
- $s = q_0$
- $B = \#$
- $F = \{q_4\}$
- δ est donné par le tableau (Tableau 9) ci-dessous (le symbole '-' indique la fonction de transition n'est pas définie pour ces valeurs).

On peut se convaincre que cette machine de Turing accepte le langage $a^n b^n$ pour $n > 0$. En effet, elle remplace de façon répétée une paire de symboles a et b respectivement par X et Y. Si tous les remplacements sont possibles et qu'une fois terminée le ruban ne contient plus aucun symbole a et b, le mot est accepté.

TABLE 9 – La matrice de transition δ

	a	b	X	Y	$\#$
q_0	(q_1, X, R)	–	–	(q_1, Y, R)	–
q_1	(q_1, a, R)	(q_2, Y, L)	–	(q_1, Y, L)	–
q_2	(q_2, a, L)	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	$(q_4, \#, R)$
q_4	–	–	–	–	–

Une machine de Turing acceptant un langage ne décrit pas toujours une procédure effective pour reconnaître ce langage. En fait, c'est la présence d'exécutions infinies qui fait qu'une machine de Turing acceptant un langage ne définit pas une procédure effective pour reconnaître ce langage. L'exécution d'une machine de Turing sur un mot w est la suite de configurations $(s, \epsilon, w)_M\, C_1 \vdash_M C_2 \vdash_M \cdots \vdash_M C_k \vdash_M \cdots$ maximale, c'est-à-dire telle que soit :

— elle est infinie

— elle se termine dans une configuration dont l'état est accepteur, ou

— elle se termine par une configuration à partir de laquelle aucune configuration n'est dérivable

Principaux résultats sur la Machine de Turing Déterministe

(i) Langage accepté par une DTM

Definition 22. — Un mot est alors accepté par une machine de Turing si l'exécution de la machine de Turing mène à une configuration dont l'état est accepteur.

— Un langage reconnu par le programme M est donné par (réponse Yes) :

$$L_M = \left\{x \in F, \delta^{(i)}(q_0, x) = (q_Y, \_, \_)\right\}$$

Nous avons donc
la définition du langage accepté par une machine de Turing.

$$L_M = \{x \in F, M \text{ accepte } x\}$$

Plus globalement, le langage L_M accepté par une machine de Turing est l'ensemble des mots w tels que :

$$(s, \epsilon, w) \vdash_M^* (p, \alpha_1, \alpha_2) \ \textbf{avec}\ p \in F \tag{6}$$

Exemple 37 : Pour l'exemple précédent illustrant la DTM, nous pouvons montrer que

$$L_M = \left\{x \in \{0,1\}^* \text{ /les deux bits les plus à droite sont à zéro}\right\}$$

(ii) Décidabilité d'une DTM

Definition 23. Un langage L est **décidé** par une machine de Turing M si

- M accepte L,
- M n'a pas d'exécution infinie

(iii) Fonction calculable par une DTM

- **Problème de Décision est résolu par cette machine**
- *Instance :* un entier positif N
- *Question :* existe-t-il un entier m tel que $N = 4.m$?

Une machine de Turing permet de calculer une fonction :

$$f_M : F \to \Gamma^*, x \in F$$

Exemple 38 : Pour l'exemple illustrant la DTM
entrée : $N = 20$ (codé par 10100)
Sortie : $N = 5$ (codé par 101)

$$f_M : \{0,1\}^* \to \{0,1,b\}^*, x \in \{0,1\}^*$$

La thèse de Turing-Church (A. Church : mathématicien qui étudia le concept de calculabilité à partir de celui de fonction) s'énonce comme suit : « Les langages reconnus par une procédure effective sont ceux décidés par une machine de Turing ». La thèse de Turing-Church identifie ainsi la notion de langage reconnaissable par une procédure effective avec celle de langage par une machine de Turing.

Exercice 39 : Construire une machine de Turing qui calcule la somme de deux nombres représentés en notation binaire.

Exercice 40 : Que peut-on dire à propos du langage accepté par une machine de Turing M si toutes les transitions de M se déplacent la tête de lecture vers la droite ?

(iv) Définition de la complexité pour une DTM

Soit M une machine de Turing Déterministe qui s'arrête toujours. La complexité en temps de M est la fonction $T_M(n)$ définie par :

$$T_M(n) = \max\{x \in F, |x| = n / M \text{ nécessite } m \text{ étapes}\}$$

Definition 24. Classe P : Un programme M est appelé programme DTM à temps polynomial s'il existe un polynôme $p(n)$ tel que $\forall n, T_M(n) \leq p(n)$

$$P = \{L : \text{ il existe un programme DTM, noté } M, \text{ tel que } L = L_M\}$$

4.4.3 Compléments : autres Machines de Turing

Différentes extensions peuvent être définies, voir par exemple :

— Machine à Ruban infini dans les deux sens (la tête de lecture peut toujours de déplacer vers la gauche)

— Machine à Rubans multiples (plusieurs rubans et plusieurs têtes de lecture). Par exemple, pour exécuter une machine à deux rubans, on utilise une machine dont l'alphabet de ruban est un ensemble de quadruplets. Deux des éléments du quadruplet sont utilisés pour représenter le contenu de chacun des rubans, les deux autres pour représenter les positions des têtes de lecture.

Exercice 41 : Proposer une machine de ruban simple qui sont équivalentes à ces deux extensions.

Machines de Turing Universelles Une question intéressante est de se demander s'il existe une machine de Turing qui peut simuler n'importe quelle machine de Turing. Précisément, nous voulons une machine de Turing quelconque M' de même qu'un mot d'entrée w et qui simulerait l'exécution de M' sur w. En termes informatiques, une telle machine de Turing serait un interpréteur de machines de Turing. De telles machines de Turing existent et sont appelées des machines de Turing universelles.

4.4.4 Machine de Turing non Déterministe (NDTM)

Schéma d'une NDTM

Définition d'une Machine de Turing non déterministe NDTM Une machine de Turing non déterministe est identique à une machine de Turing déterministe sauf en ce qui concerne la fonction de transition. Celle-ci est maintenant une relation de la forme :

$$\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\}) \tag{7}$$

c'est à dire que pour un état et une configuration donnée, Δ définit non pas un seul triplet de $(Q \times \Gamma \times \{L, R\})$, mais bien un ensemble de tels triplets. Intuitivement, lors d'une exécution, la machine peut choisir n'importe lequel de ces triplets. Formellement, cela s'exprime par le fait qu'une configuration est dérivable d'une autre où l'état est q et le symbole lu a si et seulement si elle l'est pour un des triplets (q', x, X) tels que $((q, a), (q', x, X)) \in \Delta$. Les autres notions sont alors définies exactement comme pour les DTMs. Toutefois, une machine de Turing non déterministe n'aura pas une exécution unique, et il suffira qu'une seule de ces exécutions contienne une configuration où l'état est accepteur pour que la machine accepte.

Formellement :

— $A \subseteq \Gamma$, $b \in \Gamma - A$, $q_0 \in Q$, $q_Y \in Q$, $q_N \in Q$
— $\delta : (Q - \{q_Y, q_N\}) \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$

Schéma d'une Machine de Turing non déterministe NDTM Un module de conjecture est donc proposé pour gérer l'indéterminisme (Figure 29).

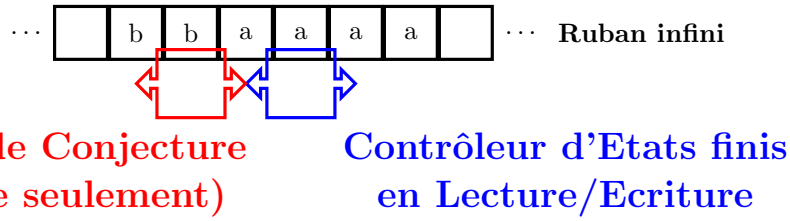


FIGURE 29 – Une Machine de Turing non Déterministe

L'instance d'entrée peut se définir par :

— $x \in F$ (symbole placé sur la case 1)
— Contrôleur de lecture/écriture pointe sur case 1
— Module de conjecture pointant sur la case −1

Principe d'une Machine de Turing non Déterministe (NDTM)

1. étape de Conjecture (module de contrôle inactif, module de conjecture actif) :
   — Déplacement du module de conjecture (case par case) vers la gauche avec écriture d'un symbole, jusqu'à arrêt éventuel (peut ne jamais s'arrêter)
2. étape de Vérification (module de contrôle actif, module de conjecture inactif) :
   — Fonctionnement DTM avec arrêt sur q_Y ou q_N

Notons que le concept de langage décidé n'a pas de sens pour les NDTM. Une NDTM a donc 3 possibilités de fonctionnement :

1. réponse Yes : « Calcul acceptable »
2. réponse No : « Calcul non acceptable »
3. ne s'arrête jamais : « Calcul non acceptable »

Quelques Résultats d'une NDTM

Définition de la complexité pour une NDTM

Definition 25. Classe NP : Un programme M est appelé programme NDTM à temps polynomial s'il existe un polynôme p tel que $\forall n, T_M(n) \leq p(n)$

$$NP = \{L : \text{ il existe un programme NDTM, noté } M, \text{ tel que } L = L_M\}$$

L'idée est alors de revenir à une DTM :

Theorem 26. *Tout langage accepté par une NDTM est aussi accepté par une DTM.*

Démonstration. Concernant la démonstration de ce théorème, il suffit que la machine de Turing déterministe simule toutes les exécutions de la machine de Turing non déterministe. Le problème lié à cette démarche est que toutes ces exécutions ne peuvent être simulées simultanément. En effet, pour simuler une exécution, nous avons besoin du ruban et celui-ci peut difficilement être utilisé par plusieurs exécutions simultanées. Une solution possible est de simuler les exécutions une par une. Le problème est alors que certaines exécutions peuvent être infinies. Si par malchance, on commence par exécuter une exécution infinie, on ne pourra jamais arriver à la simulation d'une autre exécution qui accepterait le mot. □

L'approche adoptée est de simuler toutes les exécutions, mais en considérant leurs préfixes de longueur croissante. Nous simulons donc d'abord les préfixes d'exécutions de longueur 1, puis ceux de longueur 2, puis 3, ... Dés qu'une des exécutions arrive à un état accepteur, le mot est accepté. Formalisons cette approche.

Pour une NDTM donnée, il y a un nombre maximum de choix possibles offerts par la relation de transition Δ, défini par :

$$r = max_{q \in Q, a \in \Gamma} |\{((q, a), (q', x, X)) \in \Delta\}| \tag{8}$$

Pour décrire les choix effectués dans un préfixe d'exécution de longueur m, il suffit de donner une suite de nombres inférieurs à r (si le choix indiqué ne correspond pas à une transition existante, l'exécution s'arrête sans accepter).

On peut alors construire une machine de Turing déterministe à trois rubans qui simule une machine de Turing non déterministe.

1. Le premier ruban contient le mot d'entrée et n'est pas modifié (il permet de retrouver le mot d'entrée lorsque l'on recommence une exécution).
2. Le deuxième ruban servira à contenir des séquences de nombres inférieur à r.
3. Le troisième ruban sert à à la machine déterministe pour simuler le comportement de la machine non déterministe.

La machine déterministe procède alors comme suit :

1. Sur le deuxième ruban, elle génère toutes les séquences finies de nombres inférieures à r. Ces séquences sont générées par ordre de longueur croissante.
2. Pour chacune de ces séquences, elle simule la machine non déterministe en utilisant les choix représentés par la séquence.
3. Elle s'arrête et accepte dés que la simulation d'une exécution de la machine non déterministe atteint un état accepteur.

(ii) Des problèmes indécidables ...

Nous allons envisager deux problèmes de la classe des problèmes indécidables : problèmes pour lesquels on ne connait pas d'algorithme qui réponde en un nombre fini d'étapes. Nous étudierons successivement le problème de l'arrêt, le jeu appelé BB illustrant ce problème de l'arrêt, et enfin nous rappellerons un autre problème indécidable, les équations diophantiennes.

Le problème de l'arrêt Le problème indécidable le plus connu est celui de l'arrêt (**halting problem**). Il s'agit simplement de déterminer si une machine de Turing donnée s'arrête (a une exécution finie) pour un mot d'entrée donné.

— **Instance** : P un programme

— **Question** : Le programme P s'arrête t-il ?

Exemple 39 : Le langage universel est celui composé de toutes les chaînes de caractères représentant une machine de Turing M et un mot d'entrée w tel que M accepte w.

Le langage universel LU est indécidable, défini par :

$$LU = \{\langle M, w\rangle \text{ tel que } M \text{ accepte } w\}$$

Theorem 27. *Le problème de l'arrêt est indécidable.*

Démonstration. Supposons que R est l'ensemble des langages décidables par une machine de Turing. Démontrons donc que le langage H pour le problème de l'arrêt, n'est pas dans R :

$$H = \{\langle M, w\rangle \text{ tel que } M \text{ s'arrête sur } w\}$$

Soit une instance $\langle M, w\rangle$ en utilisant une procédure de décision hypothétique pour H (approche par réduction à partir de LU).

1. Appliquer l'algorithme décidant H à $\langle M, w\rangle$.
2. Si l'algorithme décidant H donne la réponse « No » (i.e. la machine M ne s'arrête pas), répondre « No » (dans ce cas, on a effectivement $\langle M, w\rangle \notin LU$).
3. Si l'algorithme décidant H donne la réponse « Yes », simuler l'exécution de M sur w et donner la réponse obtenue (dans ce cas, l'exécution de M sur w est finie et l'on obtient toujours une réponse).

TABLE 10 – Estimation de $\Sigma(m,n)$

	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$
$m=2$	4	6	13	≥ 4098	$\geq 3.5 \times 10^{18267}$
$m=3$	9	≥ 374676383	$\geq 1.3 \times 10^{7036}$		
$m=4$	≥ 2050	$\geq 3.7 \times 10^{6518}$			
$m=5$	$\geq 1.7 \times 10^{352}$				
$m=6$	$\geq 1.9 \times 10^{4933}$				

□

Theorem 28. *Le problème de déterminer si un programme écrit dans un langage de programmation usuel s'arrête pour des valeurs fixées de ses données est indécidable.*

Exercice 42 : Démontrer que déterminer si deux machines de Turing acceptent le même langage est un problème indécidable.

Le « jeu du castor affairé » illustre le problème de l'arrêt dans un contexte particulier d'un jeu.

Exemple 40 : Le « jeu du castor affairé » (« *busy beaver game* ») introduit en 1962 par Tibor Radó [17] : Un castor affairé est une machine de Turing qui cherche à déterminer le nombre de pas effectués (ou le nombre de symboles écrits avant son arrêt). Celles-ci doivent s'arrêter après être lancées sur un ruban infini. Une fonction du castor affairé à n états n'est pas calculable. En fait, au bout d'un certain point, une fonction du castor affairé croît plus rapidement que n'importe quelle fonction calculable. Déterminer le castor affairé pour un ensemble de machines de Turing à n états donnés est un problème insoluble algorithmiquement.

Soit $\Sigma(m,n)$ est le score maximal parmi toutes les machines de Turing à m symboles et n états (de même, le nombre de pas est défini par $S(m,n)$). Il est possible de calculer ces valeurs pour 2 symboles et pour des petites valeurs de n : $\Sigma(2,0)=0$, $\Sigma(2,1)=1$, $\Sigma(2,2)=4$, $\Sigma(2,3)=6$, $\Sigma(2,4)=13$ (Cf. Table 10).

Nous supposerons dans la suite : 2 symboles $\{0,1\}$ où 0 est le symbole vierge.

Si la machine ne contient qu'un seul état (q_0) et $m=2$ symboles, le castor affairé correspond à la table de transition suivante.

À partir d'un ruban vierge, cette machine lit tout d'abord le symbole 0 : elle écrit donc le symbole 1, déplace le ruban à droite (idem à gauche) et s'arrête. Nous obtenons donc $S(2,1)=\Sigma(2,1)=1$.

$\delta(q,s)$	0	1
q_0	$(q_Y,1,+1)$	-

Si la machine ne contient que deux états (q_0,q_1) et $m = 2$ symboles, le castor affairé correspond à la table de transition suivante :
$S(2,2) = 6$, $\Sigma(2,2) = 4$ (la DTM s'arrête au bout de 6 pas, avec quatre 1 écrits sur le ruban).

$\delta(q,s)$	0	1
q_0	$(q_1, 1, +1)$	$(q_1, 1, -1)$
q_1	$(q_0, 1, -1)$	$(q_Y, 1, +1)$

Équation Diophantienne Les équations diophantiennes sont également dans la classe des problèmes indécidables. Une équation diophantienne est une équation polynomiale à une ou plusieurs inconnues dont les solutions sont cherchées parmi les nombres entiers, éventuellement rationnels, les coefficients étant eux-mêmes également entiers.

Exemple 41 : Un exemple d'équation diophantienne est l'équation $x^n + y^n = z^n$. Il a fallu attendre 1994 pour une preuve par Wiles. Ainsi, le théorème de Fermat-Wiles affirme qu'il n'existe pas de nombres entiers strictement positifs x,y, z (pour $n > 2$) vérifiant cette équation.

— **Instance** : e une équation
— **Question** : L'équation Diophantienne e a t-elle une solution ?

Theorem 29. *Une équation diophantienne est indécidable.*

4.5 Classes de problèmes : P et NP

4.5.1 Discussion sur P et NP

P vs. NP

Relation entre P et NP

Si un problème est résolu par un algorithme déterministe à temps polynomial, alors il peut être aussi résolu par un algorithme non déterministe à temps polynomial (réciproque non vérifiée)

Theorem 30. *Si $\pi \in$ NP alors il existe un polynôme p tel que π peut être résolu par un DTM en $O(2^{p(n)})$, n taille de l'entrée.*

Interprétation P et NP

Exemple 42 : « Le facteur sonne toujours deux fois » (Exemple repris de Delahaye [3] sur les ordinateurs quantiques et adaptée à notre problématique)

1. Le facteur, a réussi à se procurer le programme que la belle Amandine utilise pour décider si elle va se baigner ou si elle va à la campagne.
   — Ce programme calcule en 23 heures, à partir des données utilisées par Amandine pour faire son choix, si oui ou non Amandine va aller se baigner.

2. Il a de même réussi à se procurer le programme que Brigitte utilise pour choisir entre baignade ou promenade bucolique.
   — Il se trouve que c'est le même programme que celui d'Amandine.
3. Amandine et Brigitte ne prennent pas nécessairement la même décision, car les données qui déterminent le choix de leurs programmes peuvent êtres différentes.
4. Notre facteur possède un ordinateur aussi rapide que ces deux beautés (dommage ! Il lui en faudrait deux). Il sait que :
   — Si Amandine ou Brigitte se baignent seules, il pourra facilement les séduire
   — Il ne doit pas aller à la baignade un jour où personne ne vient ou un jour où les deux femmes se baignent simultanément, car il ne peut pas séduire deux femmes à la fois
   — Aussi il doit savoir avec certitude quand l'une des deux femmes ira se baigner et pas l'autre (il lui est indifférent de savoir laquelle).

Pour nous, « simple mortel »
A notre niveau : Avec un unique ordinateur et toutes les données obtenues des deux personnages

Nous ne pouvons pas nous en sortir puisqu'il y a le choix entre utiliser son ordinateur pour prévoir les déplacements (1) d'Amandine ou ceux (2) de Brigitte.

— Nous pouvons calculer le choix de l'un des deux mais cela ne suffit pas.
— Aucune solution classique existe pour ce problème.

Le facteur a de la chance
Les problèmes **NP** sont des problèmes qu'il peut traiter en temps polynomial.

Il fait des choix aléatoires successifs (c'est pourquoi on utilise le terme Non Déterminisme), mais à chaque fois, sa chance lui fait faire les bons choix, et il réussit donc rapidement à trouver une solution (s'il en existe une).

— jour 1 : Il décide de ne rien faire
— jour 2 : Il décide de ne rien faire
— jour 3 : Il décide d'aller à la baignade et rencontre Amandine
— jour 4 : Il décide de ne rien faire
— jour 5 : Il décide de se promener et rencontre Brigitte
— etc.

Definition 31. Un problème **NP** est dit **NP – complet** si sa résolution en temps polynomial déterministe (sans compter sur la chance) permettrait de résoudre en temps polynomial tous les autres problèmes **NP**.

Exemple 43 : Quelques problèmes **NP – complet** :

— Recherche de chemin (problème **TSP**)

— Ordonnancement

— Évaluation d'une formule logique (problème **SAT**)

— Coloration de graphes (problème **k – COL**)

4.5.2 Remarques fondamentales

A RETENIR

— Bien que les problèmes soient difficiles à trouver, une fois que nous avons une solution *yes*, nous pouvons facilement prouver que celle-ci répond au problème (souvent en une complexité linéaire).

Exemple 44 : il est difficile de trouver un circuit hamiltonien, mais facile à tester si un tel circuit est vérifié (voir plus loin)

— Nous commençons avec une solution partielle et nous l'étendons. Si nous pouvons déterminer si la prochaine étape conduit à une réponse *yes* ou *no*, nous pouvons trouver une solution globale : appelé le non déterminisme.

— Il est possible en temps polynomial de réduire un problème **NP – complet** en un autre problème **NP – complet**.

Exemple 45 : problème **TSP** peut être re-formulé comme un chemin hamiltonien (voir plus loin).

La question : P = NP ?

— Trivialement, **P** $\subseteq$ **NP** (« qui peut le plus, peut le moins »).

— **P** = **NP** signifie qu'il est possible d'énumérer toutes les solutions très rapidement (pb pour la crypto.)

— L'une des questions les plus importantes de l'informatique (question ouverte depuis 1971)

On pense aujourd'hui que tous les problèmes **NP – complet** ne peuvent pas être résolus en temps polynomial déterministe.
Si cela était possible, cela signifierait que même sans avoir de la chance, nous pourrions faire aussi bien que le facteur. En d'autre termes, il est aussi facile de trouver une solution que d'en vérifier une.

Conjecture admise :
P $\neq$ **NP** (actuellement, pas de réponse à cette question)

4.6 Transformation Polynomiale

Transformation Polynomiale : quelques éléments

4.6.1 Définition d'une Transformation Polynomiale

Definition 32. Une transformation polynomiale d'un langage $L_1 \subseteq F_1$ à un langage $L_2 \subseteq F_2$ est une fonction $f : F_1 \to F_2$ qui satisfait les deux conditions suivantes :

1. Il existe un DTM à temps polynomial qui calcule f
2. $\forall x \in F, x \in L_1$ si et seulement si $f(x) \in L_2$

On dit que L_1 est transformée en L_2 (notée $L_1 \propto L_2$)

Lemme 1
Si $L_1 \propto L_2$ alors $L_2 \in \mathbf{P} \to L_1 \in \mathbf{P}$

Exercice 43 : Démontrer le lemme 1.

Pour une approche plus approfondie, Cf. par exemple, le livre de Paschos [9].

4.6.2 Conséquence à propos de la décision

Conséquence à propos du problème initial de la décision

Definition 33. Supposons π_1 et π_2 deux problèmes de décision, avec $f : D_{\pi_1} \to D_{\pi_2}$ satisfaisant les deux conditions suivantes :

1. f est calculable par un algorithme à temps polynomial
2. $\forall I \in D_{\pi_1}, I \in Y_{\pi_1}$ si et seulement si $f(I) \in Y_{\pi_2}$

4.6.3 Rappel sur la formalisation de notre étude de cas en terme de problème de décision

Revenons au problème non résolu de la tournée du facteur :

Le problème de décision du facteur (noté π) se définit ainsi :

- **Instance :**
  - ensemble de maisons $\{M_1, M_2, \ldots, M_n\}$
  - distance $d(M_i, M_j) \in \aleph$
  - constante a
- **Question :**
  - existe t-il une tournée ?

$$\langle M_{\pi_1}, M_{\pi_2}, \ldots, M_{\pi_n}, M_{\pi_1} \rangle / \sum_{i=1}^{n-1} (d(M_{\pi_i}, M_{\pi_{i+1}})) + d(M_{\pi_n}, M_{\pi_1}) \leq a$$

4.7 Approfondissement sur les problèmes classiques de décision

4.7.1 Problème de logique

Problème SAT

Essayons d'établir la **NP**-complétude d'un problème de logique **(construction proposée par P. Wolper [11] et reprise dans ce document)**. Supposons les notations des connecteurs :$\vee$ pour le *ou logique*, $\wedge$ pour le *et logique*, $\rightarrow$ pour *l'implication logique*.

Le problème le plus connu est le « problème de satisfaisabilité » (*satisfiabilité*) pour le calcul des propositions.

— Une expression est *valide* si et seulement si elle est vraie pour toutes interprétations des propositions.

— Une expression est *satisfaisable* si et seulement s'il existe au moins une interprétation la rendant vraie

Exemple 46 : L'expression $(p \wedge (q \vee \neg r))$ n'est pas valide ; mais elle est satisfiable pour l'interprétation $\{p \rightarrow V, q \rightarrow F, r \rightarrow F\}$.

Exemple 47 : L'expression $(p \rightarrow p)$ est valide.

Dans ce qui suit, nous considérons les formes normales conjonctives (FNC).

Definition 34. une forme normale conjonctive (FNC) est définie par :

$$C_1 \wedge C_2 \wedge \cdots C_i \wedge \cdots \wedge C_m$$

où C_i (une clause) est une expression de la forme :

$$x_1 \vee x_2 \vee \cdots x_j \vee \cdots \vee x_k$$

Chaque x_j est un littéral positif ou négatif (p ou $\neg p$).

Exemple 48 : La formule $(p_1 \vee \neg p_2) \wedge p_3 \wedge (p_4 \vee p_5 \vee p_6)$ est une FNC.

Toute formule quelconque en calcul propositionnel peut s'écrire en FNC.

4.7.2 Théorème de Cook (1971)

Problème de décision SAT

Étant donné un ensemble U de n variables binaires $x_1, \cdots, x_n$ et un ensemble C de m clauses, on cherche un modèle pour la formule $\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, c'est-à-dire une affectation de valeurs booléennes $\{0, 1\}$ tel que $\mu(Phi) = 1$. Le problème de la satisfiabilité (appelé **SAT**) d'un FNC pourrait s'écrire :

— **Instance :** « une formule en FNC ».

— **Question :** « déterminer s'il existe une fonction d'interprétation μ dont l'évaluation donne vraie ».

Theorem 35. *(Théorème de Cook) Le problème* **SAT** *est* **NP – complet**. *[15]*

Remarque importante :
La construction de la preuve peut paraitre compliqué (elle se réfère au langage et au machine de Turing) ; puisque pour déterminer si un problème donné est **NP – complet**, il faut en connaitre déjà un (et donc prouver que ce prolème est **NP – complet**).

Démonstration. Pour démontrer que **SAT** est **NP – complet**, il faut établir d'une part que : (i) **SAT** est dans **NP**, et (ii) il existe une transformation polynomiale de tout problème dans **NP** vers **SAT**.

Démontrer que **SAT** $\in$ **NP** est simple, un algorithme non déterministe polynomial qui résout ce problème, est celui qui :

1. génère de façon non déterministe une fonction d'interprétation,
2. vérifie que cette fonction d'interprétation rend la formule vraie

Établissons maintenant l'existence d'une transformation polynomiale de tout langage L de **NP** vers le langage de l'encodage des instances positives de **SAT**, $L_{\mathbf{SAT}}$. Une transformation polynomiale est ici une fonction les arguments sont un mot et un langage. Supposons maintenant que nous fixons le deuxième argument, nous obtenons une transformation classique (par définition). Donc une transformation à deux arguments (mot et langage) permet de définir directement une transformation pour chaque valeur possible du langage.

Nous adopterons cette analyse pour montrer l'existence d'une transformation qui à tout mot w et tout langage $L \in$ **NP** associe une instance de $L_{\mathbf{SAT}}$ qui est positive si et seulement si $w \in L$. De plus, nous vérifierons que pour tout langage fixé, cette transformation est polynomiale en fonction de la taille du mot. Ainsi nous aurons démontré qu'il existe une transformation polynomiale de tout langage de **NP** vers $L_{\mathbf{SAT}}$.

La seule caractérisation générique des langages de **NP** est qu'ils sont acceptés par une machine de Turing non déterministe polynomiale (NDTM). Nous devons écrire une construction à partir d'une NDTM, appelé M, et d'un mot w, produit une instance de **SAT** qui est positive si et seulement si M accepte w. Soit M une NDTM dont la complexité est bornée par un polynôme $p(n)$ et $w = w_1 \cdots w_n$ un mot de Σ^*. La machine M accepte w si et seulement si il existe une exécution de M sur w de longueur au plus $p(n)$ qui mène à un état accepteur. Par simplification, nous supposerons que si la machine M atteint la configuration finale, elle poursuit son exécution en restant dans cette configuration et donc que chaque exécution comporte exactement $p(n)+1$ configurations (la machine M accepte w : une telle exécution pet être représentée par une suite d'au plus $p(n)+1$ configurations successives de M). Chaque configuration est constituée de l'état, du contenu du ruban et de la position de la tête de lecture. Notons immédiatement que le contenu non blanc du ruban a au plus la longueur $p(n)+1$, puisqu'à chaque étape de l'exécution, la tête de lecture se déplace d'une seule case.

Une exécution de M peut donc être décrite par les éléments suivants :

1. Un tableau de dimensions $(p(n)+1, p(n)+1)$ donnant pour chaque configuration et chaque case du ruban le symbole de l'alphabet du ruban Γ se trouvant dans cette case. Nous représentons l'élément i,j de ce tableau $R[i,j]$.
2. Un vecteur de dimension $p(n)+1$ donnant l'état correspondant à chaque configuration. Nous représenterons l'élément i de ce vecteur par $Q[i]$.
3. Un vecteur de dimension $p(n)+1$ donnant la position de la tête de lecture dans chaque configuration. Nous représenterons l'élément i de ce vecteur par $P[i]$.
4. Un vecteur de dimension $p(n)+1$ donnant le choix non déterministe effectué par M à chaque étape. Nous représenterons l'élément i de ce vecteur par $C[i]$. A chaque étape, la machine de Turing non déterministe M a le choix entre plusieurs transitions. Soit r le nombre maximum de ces choix. Chaque élément de C est donc un nombre compris entre 1 et r. Le vecteur C n'est pas indispensable à la description d'une exécution. Il rend simplement les choix faits par la machine de Turing non déterministe explicites. Il nous sera utile pour vérifier qu'un contenu des tableaux R, Q et P définit bien une exécution de M.

Le principe de transformation vers **SAT** est de choisir un ensemble de variables propositionnelles telles qu'une fonction d'interprétation pour ces variables définisse un contenu des tableaux R, Q, P et C. La transformation produira alors une formule qui n'est pas satisfaite que par un contenu de ces tableaux définissant une exécution M acceptant w et qui donc ne sera satisfiable que si M accepte w. Dans ce but, nous introduisons une variable propositionnelle par case des tableaux et par valeur pouvant apparaître dans ces cases. l'ensemble des variables propositionnelles est donc le suivant :

1. Une proposition $r_{ij\alpha}$ pour $0 \leq i,j \leq p(n)$ et $\alpha \in \Gamma$
2. Une proposition $q_{i\kappa}$ pour $0 \leq i \leq p(n)$ et $\kappa \in Q$
3. Une proposition p_{ij} pour $0 \leq i,j \leq p(n)$
4. Une proposition c_{ik} pour $0 \leq i \leq p(n)$ et $1 \leq k \leq r$

Notons immédiatement qu'en fonction de n (taille de w), le nombre de ces propositions est $O(p(n)^2)$ et est donc polynomial (pour une machine de Turing fixée, $|Gamma|$, $|Q|$ et r sont des constantes). On peut considérer qu'une fonction d'interprétation pour ces propositions définit un contenu des tableaux en adoptant la convention que $r_{ij\alpha} \leftarrow 1$ signifie que le symbole se trouvant dans la case $R[i,j]$ est α, que $q_{i\kappa} \leftarrow 1$ signifie que l'état représenté dans $Q[i]$ est κ, ... Toutefois cette convention n'a de sens que s'il existe un seul $\alpha \in \Gamma$ tel que $r_{ij\alpha} \leftarrow 1$, un seul $\kappa \in Q$ tel que $q_{i\kappa} \leftarrow 1$, ... Il faut donc restreindre les fonctions d'interprétation satisfaisant une formule exprimant la condition correspondante. Pour les variables propositionnelles $r_{ij\alpha}$, cette formule est :

$$\bigwedge_{0 \leq i,j \leq p(n)} \left[\bigvee_{\alpha \in \Gamma} r_{ij\alpha} \wedge \bigwedge_{\alpha \neq \alpha' \in \Gamma} \left(\neg r_{ij\alpha} \vee \neg r_{ij\alpha'} \right) \right] \tag{9}$$

Une formule semblable existe pour les propositions $q_{i\kappa}$ et c_{ik} (dans la suite, nous mentionnerons uniquement eq. 9) pour faire référence à l'ensemble de ces formules). Notons que la formule 9 est en forme normale conjonctive et que sa longueur est $O(p(n)^2)$. Pour les propositions p_{ij}, la même condition s'exprime par la formule 10, qui est de longueur $O(p(n)^3)$.

$$\bigwedge_{0\leq i\leq p(n)}\left[\bigvee_{0\leq j\leq p(n)} p_{ij} \wedge \bigwedge_{j\neq j'}(\neg p_{ij} \vee \neg p_{ij'})\right] \tag{10}$$

Donc, toute fonction d'interprétation qui satisfait les formules 9 et 10 définit un contenu des tableaux R, Q, P et C. Nous allons maintenant prendre la conjonction de 9 et 10 , avec d'autres formules qui imposeront que toute fonction d'interprétation qui les satisfait définit une exécution de M acceptant w. Trois conditions doivent être imposées :

1. La première configuration est la configuration initiale
2. Chaque configuration est obtenue à partir de la précédente suivant la relation de transition de la machine de Turing
3. Une configuration finale apparait au plus tard après $p(n)$ étapes.

Pour que la première configuration soit initiale, il faut que le mot w figure sur le ruban (le reste du ruban contenant le symbole *blanc*, noté b), que la tête de la lecture soit sur la première case du ruban et que l'état soit l'état initial. Si $w = w_1 \cdots w_n$, ces conditions sont exprimées par la formule 11, qui est une FNC de longueur $O(p(n))$.

$$\left[\bigwedge_{0\leq j\leq n-1} r_{0jw_{j+1}} \wedge \bigwedge_{n\leq j\leq p(n)} r_{0jb}\right] \wedge q_{0s} \wedge p_{0O} \tag{11}$$

Il y a deux conditions qui expriment la relation entre la configuration et la suivante :

1. Toutes les cases du ruban ne se trouvant pas sous la tête de lecture ne sont pas modifiées
2. La case se trouvant sous la tête de lecture, l'état et la position de la tête de la lecture sont modifiées en concordance avec la relation de transition.

La formule 12 représente la première condition. Elle exprime que si, dans la configuration i, le symbole de la case j est α et que la tête de lecture ne se trouve pas sur cette case, alors, dans la configuration $i+1$, le symbole se trouvant dans la case j est aussi α.

$$\bigwedge_{0\leq i,j\leq p(n),\alpha\in\Gamma}\left[(r_{ij\alpha} \wedge \neg p_{ij}) \rightarrow r_{(i+1)j\alpha}\right] \tag{12}$$

Cette formule peut se réécrire en FNC comme suit :

$$\bigwedge_{0\leq i,j\leq p(n),\alpha\in\Gamma}\left[\neg r_{ij\alpha} \vee p_{ij} \vee r_{(i+1)j\alpha}\right] \tag{13}$$

Grâce aux propositions c_{ik}, la condition exprimant la relation entre configurations successives est relativement simple à exprimer. EN effet, supposons que dans la configuration i, l'état soit κ, la position de la tête de lecture soit j, le symbole lu soit α et le choix non déterministe soit k. Pour ces éléments, la relation de transition Δ de M nous précise de façon unique l'état suivant $\kappa^{'}$, le symbole à écrire sur le ruban $\alpha^{'}$ et le déplacement d de la tête de lecture (+1 ou −1). La contrainte imposée par Δ est alors exprimée par eq. 14.

$$\bigwedge_{0\leq i,j\leq p(n),\alpha\epsilon\Gamma,\kappa\epsilon Q,1\leq k\leq r}\begin{bmatrix}\big((q_{ik}\wedge p_{ij}\wedge r_{ij\alpha}\wedge c_{ik})\rightarrow q_{(i+1)\kappa'}\big)\wedge\\ \big((q_{ik}\wedge p_{ij}\wedge r_{ij\alpha}\wedge c_{ik})\rightarrow r_{(i+1)j\alpha'}\big)\wedge\\ \big((q_{ik}\wedge p_{ij}\wedge r_{ij\alpha}\wedge c_{ik})\rightarrow p_{(i+1)(j+d)}\big)\end{bmatrix} \tag{14}$$

ou sous forme normale conjonctive par eq. 15.

$$\bigwedge_{0\leq i,j\leq p(n),\alpha\epsilon\Gamma,\kappa\epsilon Q,1\leq k\leq r}\begin{bmatrix}\big(\neg q_{ik}\vee\neg p_{ij}\vee\neg r_{ij\alpha}\vee\neg c_{ik}\vee q_{(i+1)\kappa'}\big)\wedge\\ \big(\neg q_{ik}\vee\neg p_{ij}\vee\neg r_{ij\alpha}\vee\neg c_{ik}\vee r_{(i+1)j\alpha'}\big)\wedge\\ \big(\neg q_{ik}\vee\neg p_{ij}\vee\neg r_{ij\alpha}\vee\neg c_{ik}\vee p_{(i+1)(j+d)}\big)\end{bmatrix} \tag{15}$$

Il est immédiat de vérifier que les formules 13 et 15 ont une longueur $O(p(n)^2)$. Il nous reste à exprimer qu'un état accepteur est atteint durant l'exécution :

$$\bigvee_{0\leq i\leq p(n),\kappa\epsilon F}[q_{i\kappa}] \tag{16}$$

En résumé, la conjonction des équations (9), (10), (11), (13), (15) et (16) est satisfaisable si et seulement la machine de Turing M accepte w. En effet, d'une part, toute fonction d'interprétation satisfaisant cette formule décrit une exécution de M acceptant w et d'autre part, une exécution de M acceptant w peut être vue comme une fonction d'interprétation satisfaisant la formule. Remarquons que les choix possibles dans l'exécution non déterministe de M sont représentés par le choix des valeurs de vérité affectés aux propositions c_{ik} par la fonction d'interprétation. De plus, la longueur de la formule est $O(p(n)^3)$ et elle peut être construite en un temps polynomial en n. Notons finalement que construire l'encodage de la formule ne nécessiterait aussi qu'un temps polynomial en fonction de sa longueur. Nous avons donc établi l'existence d'une transformation polynomiale de tout langage L de **NP** vers $L_{\mathbf{SAT}}$. □

La **NP**-complétude de **SAT** est établie, il est possible de montrer par transformation que d'autres problèmes sont **NP – complets**.

4.7.3 Autres problèmes NP – complets : 3 – SAT, k – COL, VC

Problème 3 – SAT

Nous traitons d'abord une restriction de **SAT** : Le problème **3 – SAT** est celui de la satisfaisabilité des formules en FNC comportant exactement 3 littéraux par clause.

Theorem 36. *Le problème* **3 – SAT** *est* **NP – complet**.

Démonstration. Démontrons que **3 – SAT** est **NP – complet**. Il est dans **NP** puisqu'il s'agit d'un cas particulier de **SAT**. Pour démontrer qu'il est **NP**-dur, nous allons établir que **SAT** $\propto$ **3 – SAT**. La transformation traite une instance de **SAT** en substituant aux clauses ne comportant pas 3 littéraux une conjonction de clauses de 3 littéraux. Nous allons décrire successivement les cas de clauses comportant 2 littéraux, un seul littéral et finalement 4 littéraux ou plus.

(1) Une clause $(x_1 \vee x_2)$ comportant deux littéraux est remplacée par :

$$(x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \neg y) \tag{17}$$

où y est une nouvelle variable propositionnelle. Toute fonction d'interprétation qui rend vraie $(x_1 \vee x_2)$ rend aussi vraie l'expression 17. inversement, une fonction d'interprétation pour laquelle eq. 17 est vraie doit affecter à x_1 ou à x_2 la valeur 1 puisque y et ${}_n egy$ ne peuvent pas être tous deux vrais. Par conséquent eq. 17 est satisfaisable si et seulement si $(x_1 \vee x_2)$ est satisfaisable.

(2) Une clause x_1 comportant un seul littéral est remplacée par :

$$(x_1 \vee y_1 \vee y_2) \wedge (x_1 \vee y_1 \vee \neg y_2) \wedge (x_1 \vee \neg y_1 \vee y_2) \wedge (x_1 \vee \neg y_1 \vee \neg y_2) \tag{18}$$

où y_1 et y_2 sont de nouvelles variables propositionnelles. Un raisonnement semblable à celui du cas précédent permet de montrer que l'équation 18 est satisfaisable si et seulement si x_1 est satisfaisable.

(3) Soit Une clause (eq. 19) comportant $l \geq 4$ littéraux est remplacée par (eq. 20), où $y_1, \cdots, y_{l-3}$ sont des variables propositionnelles.

$$(x_1 \vee x_2 \vee \cdots \vee x_i \vee \cdots \vee x_{l-1} \vee x_l) \tag{19}$$

$$\begin{gathered}(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \cdots \wedge \\ (\neg y_{i-2} \vee x_i \vee y_{i-1}) \wedge \cdots \wedge (\neg y_{l-4} \vee x_{l-2} \vee y_{l-3}) \wedge (\neg y_{l-3} \vee x_{l-1} \vee x_l)\end{gathered} \tag{20}$$

La clause (19) est satisfaisable si et seulement si la formule (20) est satisfaisable. En effet, si une fonction d'interprétation rend (19) vraie, elle doit affecter la valeur 1 à au moins un des littéraux de cette clause. Supposons que ce littéral soit x_i. Dans ce cas, il suffit d'étendre la fonction d'interprétation en affectant 1 à tous les y_i tels que $j \leq i-2$ et en affectant 0 à tous les y_i tels que $j \geq i-1$ pour obtenir une fonction d'interprétation rendant (20) vraie.

Inversement, si une fonction d'interprétation rend (20) vraie, elle doit affecter la valeur 1 à au moins un des x_i. En effet, supposons que tous les x_i aient la valeur 0. Il faut alors que $y_1 \leftarrow 1$ pour que la première clause de (20) soit vraie. Poursuivant le raisonnement, il faut que $y_2 \leftarrow 1$ pour que la deuxième clause soit vraie, ..., et que $y_{l-3} \leftarrow 1$ pour que le pénultième clause soit vraie. Mais, dans ce cas, la dernière clause ne peut être vraie. Une contradiction.

La construction que nous venons de décrire est aisément implémentée par un algorithme polynomial. Nous avons donc démontré que **3 – SAT** est un problème **NP – complet**.

□

Rappelons que des petites modifications peuvent changer la classe. Par exemple, modifions ce problème, au lieu de considérer **3 – SAT**, étudions **2 – SAT**.

Exercice 44 : Montrer que **2 – SAT** est de classe P, alors que nous venons juste de prouver que **3 – SAT** est **NP**.

Problème VC

Le problème de la couverture de sommets **VC** (*vertex cover*) a pour données un graphe et une constante j. Il consiste à déterminer s'il existe un sous-ensemble des sommets du graphe de taille au plus j qui couvre tous les arcs du graphe, à savoir tel que chaque arc a au moins une de ses extrémités dans l'ensemble.

Formellement, étant donnés un graphe $G = (V, E)$ et un entier $j \leq |V|$, il faut déterminer s'il existe un sous-ensemble $V' \subseteq V$ tel que $|V'| \leq j$ et tel que pour tout arc $(u, v) \in E$, avec $u, v \in V'$.

Theorem 37. **VC** *est* **NP – complet**.

Démonstration. Il est aisé de voir que **VC** est dans **NP**. En effet, l'algorithme qui :

— génère un ensemble de sommets de façon non déterministe

— vérifie que cet ensemble de sommets est une couverture de graphe

est un algorithme non déterministe polynomial qui résout **VC**.

Une transformation polynomiale de **3 – SAT** vers **VC** permet de démontrer que **VC** est **NP**-difficile (ou **NP**-dur). Soit $E_1 \wedge E_2 \wedge \cdots \wedge E_k$ une instance de **3 – SAT**. Chaque E_i est de la forme $x_{i1} \vee x_{i2} \vee x_{i3}$ où x_{ij} est un littéral. Supposons que l'ensemble des variables propositionnelles apparaissant dans l'instance est $\mathcal{P} = \{p_1, p_2, \cdots, p_l\}$.

L'instance de **VC** qui est construite est alors la suivante.

(1) L'ensemble V des sommets du graphe contient :

(a) une paire de sommets étiquetés p_i et $\neg p_i$ pour chaque variable propositionnelle de $\mathcal{P}$,

(b) un triplet de sommets du graphe étiquetés x_{i1},x_{i2}, x_{i3}

Le nombre de sommets du graphe est donc égal à $2l + 3k$.

(2) L'ensemble E des arcs du graphe contient :

(a) l'arc $(p_i, \neg p_i)$ pour chaque paire de sommets p_i, $\neg p_i$, $1 \leq i \leq l$

(b) les arcs (x_{i1}, x_{i2}), (x_{i2}, x_{i3}) et (x_{i3}, x_{i1}) pour chaque triplet de sommets x_{i1},x_{i2}, x_{i3}, $1 \leq i \leq k$

(c) un arc entre chaque sommet x_{ij} et le sommet p ou $\neg p$ représentant le littéral correspondant

Le nombre d'arcs du graphe est donc égal à $l + 6k$.

(3) La constante j est $l + 2k$.

La transformation que nous venons de décrire est visiblement polynomiale. Montrons que le graphe généré a une couverture de sommets de taille au plus j si et seulement si l'instance de $\mathbf{3-SAT}$ est satisfaisable. Supposons qe l'instance de $\mathbf{3-SAT}$ soit satisfaisable. Dans ce cas, il existe une fonction d'interprétation μ qui rend chaque clause E_i vraie. Nous définissons une couverture de sommets du graphe à partir de cette fonction d'interprétation. La couverture comprend les sommets suivants :

1. les sommets p_i tels que μ affecte 1 à p_i, et les sommets $\neg p_i$ tels que μ affecte 0 à p_i.
2. deux sommets parmi chaque groupe de trois sommets x_{i1},x_{i2}, x_{i3} choisis de telle façon que la fonction d'interprétation affecte la valeur 1 au littéral correspondant au sommet non choisi.

Cette couverture a la taille $l + 2k$. Vérifions qu'elle couvre bien tous les arcs du graphe.

1. Les arcs $(p_i, \neg p_i)$ sont couverts puisque soit p_i, soit $\neg p_i$ est dans la couverture
2. les arcs (x_{i1}, x_{i2}), (x_{i2}, x_{i3}) et (x_{i3}, x_{i1}) sont couverts puisque deux des trois sommets x_{i1},x_{i2}, x_{i3} sont dans la couverture
3. Les arcs entre chaque sommet x_{ij} est le sommet p ou $\neg p$ représentant le littéral correspondant sont couverts puisque soit x_{ij} est dans la couverture, soit x_{ij} n'est pas dans la couverture mais correspond à un littéral rendu vrai par la fonction d'interprétation et donc le sommet p ou $\neg p$ se trouvant à l'autre extrémité de l'arc est dans la couverture.

Inversement, si le graphe a une couverture de taille $l + 2k$, celle-ci définit une fonction d'interprétation qui satisfait l'instance de $\mathbf{3-SAT}$. En effet, une couverture de taille $l + 2k$ doit comprendre un sommet parmi chaque triplet x_{i1},x_{i2}, x_{i3} et ne peut comprendre d'autres sommets (sinon la taille serait supérieure à $l + 2k$). Considérons la fonction d'interprétation telle que $p_i \leftarrow 1$ si p_i est dans la couverture et $p_i \leftarrow 0$ si $\neg p_i$ est dans la couverture. L'existence de la couverture implique que cette fonction d'interprétation satisfait l'instance de $\mathbf{3-SAT}$. En effet, pour que tous les arcs reliant les sommets x_{ij} aux sommets p et $\neg p$ soient couverts, il faut qu'un de ces arcs soit couvert du coté des sommets p, $\neg p$ pour chaque triplet de sommets x_{i1},x_{i2}, x_{i3}. Cela implique que la fonction d'interprétation affecte la valeur 1 à au moins un littéral de chaque clause et donc qu'elle satisfait la formule.

Nous pouvons donc conclure que le problème $\mathbf{VC}$ est $\mathbf{NP-complet}$. □

Problèmes de coloration de graphe

Le problème de coloration (appelé $\mathbf{k-COL}$) de graphe consiste à déterminer si k couleurs suffisent pour colorier les points d'un graphe de sorte que deux points reliés par un arc n'aient jamais la même couleur.

— **Instance :** G : un graphe, k couleurs
— **Question :** G peut-il être colorié par k couleurs ?

Theorem 38. **k – COL** *est* **NP – complet**.

Theorem 39. **3 – COL** *est* **NP – complet**.

Démonstration. On le montre en réduisant **3 – SAT** en **3 – COL**. □

Problème SUM

Definition 40. Le *problème* **SUM** est défini par :
— **Instance** : Un ensemble E de n entiers ; S un
— **Question** : Existe-t-il des éléments de E dont la somme donne S ?

Theorem 41. **SUM** *est* **NP – complet**.

Focus sur deux autres problèmes TSP et HC

Definition 42. Le *Voyageur de Commerce* (**TSP** *problem*) est défini par :
— **Instance**
  — ensemble de villes $\{v_1, v_2, \ldots, v_n\}$
  — distance $d'(v_i, v_j) \in \aleph$
  — constante a'
— **Question**
  — existe t-il une tournée ?

$$\langle v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_n}, v_{\pi_1} \rangle / \sum_{i=1}^{n-1} (d'(v_{\pi_i}, v_{\pi_{i+1}})) + d'(v_{\pi_n}, v_{\pi_1}) \leq a'$$

Theorem 43. **TSP** *est* **NP – complet**.

Le problème de circuit hamiltonien (**HC**) est aussi **NP – complet**.

Definition 44. Un Circuit Hamiltonien (appelé **HC** pour *Hamiltonian path* ou *Hamiltonian cycle*) est défini par un circuit incluant tous les sommets de G :
— **Instance** : un graphe $G(V, E)$; $o, d \in V$
— **Question** : existe-t-il un circuit hamiltonien pour G de o à d ?

$$\langle v_1, v_2, \ldots, v_n \rangle, v_i \in V / \forall 1 \leq i \leq n, \{v_i, v_{i+1}\} \in E, \{v_n, v_1\} \in E$$

Theorem 45. **HC** *est* **NP – complet**.

Démonstration. Pour **HC**, cela se démontre par une démonstration de **TSP** vers **HC**. Il est aussi possible de décrire une transformation de **HC** vers **TSP**, qui permet d'établir la **NP**-complétude.

(i) Proposition d'une fonction de transformation Le problème consiste à définir une fonction de transformation f :

$$\overset{\textbf{HC}}{G(V_1, E_1), |V| = m} \xrightarrow{f} \overset{\textbf{TSP}}{G(V_2, E_2), d'(v_i, v_{i+1}), a'}$$

— $V_2 = V_1$,

— si $\{x, y\} \in V_1$ alors $d'(x, y) = 1$ sinon $d'(x, y) = 2$,

— $a' = m$

(ii) Vérification des deux conditions pour f

1. Il existe un DTM à temps polynomial qui calcule f ? Taille de l'entrée : un polynôme en fonction de G_1 de **HC**
   $f(|V|) \leq$ nombre d'arêtes de $G_2 = \frac{m.(m-1)}{2}$
2. $\forall x \in F^*, x \in L_1$ si et seulement si $f(x) \in L_2$?
   — $x \in L_1$, i.e. $\langle v_1, v_2, \ldots, v_m, v_1\rangle$ chemin hamiltonien alors $d'(\langle v_1, v_2, \ldots, v_m, v_1\rangle) = m \leq a'$
   donc $f(x) \in L_2$
   — $f(x) \in L_2$, i.e. chercher $t = \langle v_1, v_2, \ldots, v_m, v_1\rangle / d'(t) \leq a'$
   implique $t = d'(\langle v_1, v_2, \ldots, v_m, v_1\rangle)$ et $d'(v_i, v_{i+1}) = 1$
   donc $x \in L_1$
   cqfd

Conclusion : **HC** $\propto$ **TSP**

Signification : « Si **TSP** peut être résolu par un algorithme à temps polynomial, alors il en est de même pour **HC**(si **HC** est indécidable, alors **TSP** l'est également) ». □

Definition 46. Si L_1 et L_2 sont polynomialement équivalents si $L_1 \propto L_2$ et $L_2 \propto L_1$

Exercice 45 : Compléter la preuve précédente en montrant que **HC**et **TSP** sont polynomialement équivalents

Rappelons que des petites modifications peuvent changer la classe. Par exemple, modifions ce problème, au lieu de considérer **HC**, étudions **eulerien**.

Exercice 46 : Montrer que **eulerien** est de classe P, alors que nous venons juste de prouver que **HC** est **NP**.

4.7.4 Qu'en est il de notre application ?

Revenons à notre étude de cas ...

Exemple 49 : La tournée de facteur est-il **NP – complet** ?

Démonstration. Le problème consiste à définir une fonction de transformation f telle que :

$$\overset{\textbf{TF}}{G(V_1, E_1), |V| = a} \xrightarrow{f} \overset{\textbf{TSP}}{G(V_2, E_2), d'(v_i, v_{i+1}), a'}$$

— $V_2 = V_1$,

— si $\{x, y\} \in V_1$ alors $d'(x, y) = d(x, y)$

— $a' = a$

Il est facile de montrer que les problèmes sont identiques.
Conclusion : **TF** $\propto$ **TSP** □

Exercice 47 : Montrer que **TF**et **TSP**sont polynomialement équivalents

Lemme 2
Si $L_1 \propto L_2$ et $L_2 \propto L_3$ alors $L_1 \propto L_3$

Exemple 50 : A partir de **TF** $\propto$ **TSP** et **TSP** $\propto$ **HC**, Nous pouvons en déduire que $TF \propto CH$.

Exemple 51 : Décision sur le problème de Distribution de courrier : Le problème de distribution de courrier conduit à des problèmes de voyageur de commerce ainsi que de circuit hamiltonien, et est considéré comme aussi difficile.

Compléments sur les Transformations polynomiales

Definition 47. Définition de la **NP**-complétude

$$L \text{ est } \textbf{NP} - \textbf{complet} \text{ si } L \in \textbf{NP} \text{ et } \forall L' \in \textbf{NP}, L' \propto L$$

Lemme 3
Si $L_1, L_2 \in \textbf{NP}, L_1$ est **NP – complet** et $L_1 \propto L_2$ alors $L_2 \in \textbf{NP} - \textbf{complet}$

4.8 Pour aller plus loin ...

Les progrès technologiques vont-ils changer la situation en ce qui concerne la complexité ? les ordinateurs parallèles se répandent, permettent-ils de vaincre la complexité ? Qu'en est-il de l'informatique quantique ? (voir le livre de Ridoux et Lesventes [10]).

4.8.1 NP-complétude vs. loi de Moore

En 1965, Gordon Moore observe que le nombre de circuits intégrés par unité de surface double tous les ans depuis 1958 (date de l'invention des circuits intégrés). Il prédit qu cela va encore durer une dizaine d'années. On appelle cette loi empirique, la **loi de Moore**. En 1975, un ajustement de la loi de Moore énonce que les performances ont doublé tous les 18 mois, et prédit que cela va durer. Autrement dit, les performances sont multipliées par 1000 en 15 ans (1000000 en 30 ans).

Cette loi n'est pas encore démentie, mais il est prévisible que la technologie ne pourra pas suivre cette loi longtemps. Une autre loi empirique, la loi de Rock, pourrait contre-balancer la loi de Moore ; puisqu'elle énonce que les investissements nécessaires au développement d'une nouvelle génération de circuits intégrés doublent tous les quatre ans.

Même en supposant que ce doublement des performances continue à être vérifié, est ce que nous pouvons suggérer d'attendre un peu pour résoudre un problème qui demande aujourd'hui trop de temps de calcul ?

Admettons que ce doublement des performances passe complètement dans la vitesse de calcul ; si la résolution d'un problème de taille s demande $f(s)$ unité de temps, sa résolution après doublement des performances en demandera $f(s)/2$. Choisissons une fonction pour laquelle la plus grande taille d'entrée qui puisse être traitée avec un temps de calcul tolérable soit S ($f(S) = \textit{max-tolérable}$). Supposons que le seuil de tolérance ne change pas dans 18 mois, quel accroissement de S sera offert par le doublement des performances (S'/S tel que $f(S')/2 = \textit{max-tolérable}$) ? Il suffit de résoudre $f(S')/2 = f(S)$ en S'. Nous pouvons en déduire $f(S') = 2 \cdot f(S)$, nous conduisant à $S' = f^{-1}(2 \cdot f(S))$ en supposant f inversible. Nous pouvons conclure par un accroissement des performances matérielles (eq. 21) :

$$\frac{S'}{S} = \frac{f^{-1}(2 \cdot f(S))}{S} \tag{21}$$

Ce résultat montre que l'accroissement dépend de la fonction f. Le tableau (Tableau 11) donne les formules d'accroissement pour quelqes ordres de grandeur. Pour toutes les complexités polynomiales, le taux d'accroissement avec le doublement des performances est une constante. Pour une complexité de n^10, l'accroissement net dépasse 7%. Pour une complexité exponentielle, l'accroissement net baisse avec la taille du problème à résoudre et tend vers zéro. **La différence entre complexité polynomiale et complexité exponentielle est à souligner : La première bénéficie du doublement des performances apportées par les progrès technologiques, tandis que la seconde n'en bénéficie presque pas, voire pas du tout.** il est donc vain d'attendre que les calculateurs s'améliorent pour exécuter certains programmes.

TABLE 11 – Formules d'accroissement de la performances suivant la loi de Moore pour différentes ordres de grandeur

Fonctions	Taux d'accroissement des performances en 18 mois
$log(n)$	e^2 (≈ 7.39)
n	2
$2 \cdot n$	2
n^2	$\sqrt{2}$ (≈ 1.414)
$n^{2.81}$	$2^{1/2.81}$ (≈ 1.279)
n^3	$2^{1/3}$ (≈ 1.26)
n^{10}	$2^{1/10}$ (≈ 1.072)
2^n	$1 + \frac{1}{n}$

4.8.2 NP-complétude vs. Parallélisme

En ce qui concerne le parallélisme, même s'il existe maintenant des machines comportant des milliers de processeurs, il ne faut jamais oublier que l'on dispose de n processeurs, on peut au plus réduire le temps de calcul d'un facteur n. Donc pour un algorithme ayant un comportement exponentiel, il faudrait disposer d'un nombre de processeurs exponentiel en fonction de la taille de l'instance à traiter pour que cet algorithme devient exploitable. **Ce n'est évidemment pas réaliste.**

Même pour des problèmes dont la complexité est basse, on peut se demander s'il est toujours possible d'exploiter une machine parallèle pour rendre la résolution plus rapide. **La réponse est en fait négative**.La question a été étudiée dans le cadre des problèmes appartenant à la classe P pour lesquels on cherche à savoir si, disposant d'un nombre polynomial de processeurs, il est possible de trouver un algorithme de complexité $O(log^c(n))$. Autrement dit, on souhaite obtenir une complexité polynomiale en $log(n)$ à l'aide d'un nombre de processeurs polynomial en n. Ce concept a été défini en termes d'une classe de complexité.

Definition 48. NC est la classe des langages qui peut être reconnue en temps $O(log^c(n))$ sur une machine comportant $O(n^k)$ processeurs

Remarquons que notre modèle de machine pour définir cette classe ne peut plus être une machine de Turing. La question de savoir si tous les problèmes de la classe **P** sont-ils dans NC ? Les problèmes les plus difficiles de la classe **P** ne sont vraisemblablement pas dans la classe NC, et ne sont donc pas parallélisables. Ces problèmes sont appelés **P-complets** et sont définis comme les problèmes **NP – complets**, mais à partir de la classe **P** et d'une notion de transformation plus restrictive que les transformations polynomiales, à savoir la *transformation espace logarithmique*. Une *transformation espace logarithmique* est définie exactement comme une transformation polynomiale, à cela près que la fonction définissant la transformation doit être calculable en utilisant un espace logarithmique.

4.8.3 NP-complétude vs. Informatique quantique

L'idée de l'ordinateur quantique est d'exploiter une particularité du monde quantique : la superposition d'états. Un ordinateur quantique plutôt que d'utiliser des bits classiques (états classiques 0 et 1), utilise des bits quantique (quantum bits), ou encore **qubits**, qui peuvent être dans un état 0, un état 1 ou une superposition de ces états. La superposition fait qu'un *qubit* représente plus d'une valeur et donc qu'un registre de n *qubits* permet d'encoder un nombre exponentiel d'états. Des mécanismes quantiques permettent alors de calculer simultanément avec ces états ce qui offre des perspectives intéressantes. Toutefois, les limites des mécanismes disponibles font que trouver des algorithmes qui exploitent les propriétés de *qubits* n'est pas simple. Une découverte importante dans le domaine est qu'il est possible de factoriser efficacement un nombre de n bits à l'aide d'un ordinateur quantique de $2n$ bits, alors que sur les machines classiques tous les algorithmes de factorisation nécessitent un temps exponentiel.

La réalisation d'ordinateurs quantiques en est encore au stade des machines expérimentales comportant quelques *qubits*, mais il est clair que si on arrive à construire des ordinateurs quantiques pratiques de capacité suffisante, par exemple, la difficulté de factorisation qui est à la base des systèmes cryptographiques ne seraient plus viables.

4.8.4 Quelques exercices supplémentaires ...

Exercice 48 : Logiques Booléennes

Notons p_i une variable booléenne, un littéral est désigné par p_i ou $\neg p_i$. Une clause est une disjonction de littéraux, $p_i \vee \neg p_j$

Un problème de décision π est défini par : Instance : Un ensemble U de variables booléennes. Un ensemble C de conjonction de clauses défini par : $C = \{c_1, c_2, \ldots, c_n\} / |c_i| = m, 1 \leq i \leq n$. Par exemple nous pouvons avoir $C = \{p_1 \vee \neg p_2, \neg p_1 \vee p_2\}$.

Question : Existe-t-il une assignation de variables booléennes satisfaisant C ? (pour notre exemple, si p_1 et p_2 sont à valeur vraie, C est satisfait)

1. Montrer qu'un problème 2-SAT (i.e. des clauses dont le nombre de variables est $m = 2$) est résolu en polynomial (classe P).
2. Montrer que 3-SAT est NP-Complet ? (Le théorème de Cook - 1971 - montre que le problème SAT pour satisfiabilité, est NP-complet)
3. Le problème 3-SAT* est un cas particulier de 3-SAT ou dans chaque clause, on a que des littéraux positifs ou négatifs. Montrer que 3-SAT* est NP-complet ?

Exercice 49 : Montrer que l'encodage pour le problème du voyageur de commerce est identique à celui proposé dans le cours par le facteur pour sa tournée, i.e. $longueur(I) = m + \lceil \log_2(a) \rceil + \max \{ \lceil \log_2(d(c_i, c_j)) \rceil, \forall c_i, c_j \}$

4.9 Conclusion

Conclusion

Prouver qu'un problème est **NP – complet** ?

Prouver qu'un problème de décision π est **NP – complet** :

1. $\pi \in$ **NP**
2. Sélectionner un problème **NP – complet**connu ; π'
3. Construire une transformation $f : \pi \to \pi'$
4. Montrer que f est une transformation polynomiale

3 classes de Problèmes ?

1. classe **P** : problèmes résolubles en temps polynomial
2. classe **NP** : problèmes vérifiables en temps polynomial
   (Nous pouvons vérifier que la solution est correcte dans un temps polynomial
3. classe **NP – complet** (problème dit **NPC**) :
   Si problème appartient à **NP**, et « il est aussi difficile que n'importe qu'elle autre problème **NP** ».

Conjecture **P** $\neq$ **NP** Une représentation très simplifié du Monde des **NP**(Figure 30).

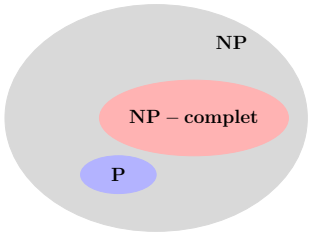


FIGURE 30 – Le Monde des **NP**

Autres classes de problème ...

Soit f une fonction $\aleph \to \Re^+$.

Definition 49. **TIME**$(f(n))$ est l'ensemble des problèmes décidés par une machine de Turing déterministe en temps $O(f(n))$.

$$\mathbf{P} = \bigcup_{k \geq 0} \mathbf{TIME}\left(n^k\right)$$

Definition 50. La classe **NTIME**$(f(n))$ est l'ensemble des problèmes décidés par une machine de Turing non déterministe en temps $O(f(n))$.

$$\mathbf{NP} = \bigcup_{k \geq 0} \mathbf{NTIME}\left(n^k\right)$$

Definition 51. La classe **EXPTIME** $(f(n))$ est l'ensemble des problèmes décidés par une machine de Turing déterministe dont la complexité temps est bornée par une fonction exponentielle ($2^{p(n)}$ où $p(n)$ est un polynôme).

$$\textbf{EXPTIME} = \bigcup_{k \geq 0} \textbf{TIME}\left(2^{n^k}\right)$$

Definition 52. **PSPACE** = **NPSPACE**, avec :

- **PSPACE** est la classe des langages décidés par une machine de Turing déterministe dont la complexité en espace (nombre de cases du ruban) est bornée par un polynôme.
- **NPSPACE** est la classe des langages décidés par une machine de Turing non déterministe dont la complexité en espace (nombre de cases du ruban) est bornée par un polynôme.

Par définition, il est admis :

Definition 53.

$$\textbf{P} \subseteq \textbf{NP} \subseteq \textbf{PSPACE} \subseteq \textbf{EXPTIME}$$

L'inclusion entre **P** et **EXPTIME** est défini, mais à l'heure actuelle, nous ne savons pas à quel niveau cette inclusion est définie (entre **P** et **NP**, entre **NP** et **PSPACE**, etc.).

A chaque situation ... Sa réponse
(reproduction du livre de Garey et Johnson [6])

situation 1 : « Argumentation A éviter »
(voir Figure 31)

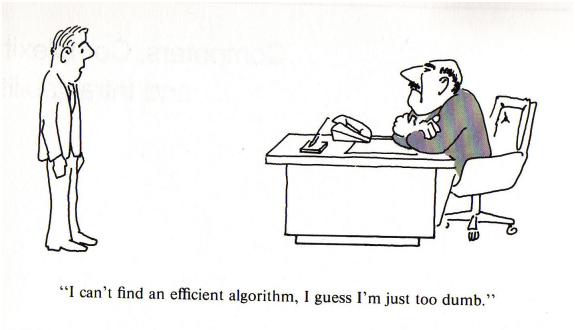


FIGURE 31 – Situation critiquable ! (figure repris de [6])

situation 2 : « Argumentation raisonnable »
(voir Figure 32)

situation 3 : « argumentation la plus sérieuse »
(voir Figure 33)

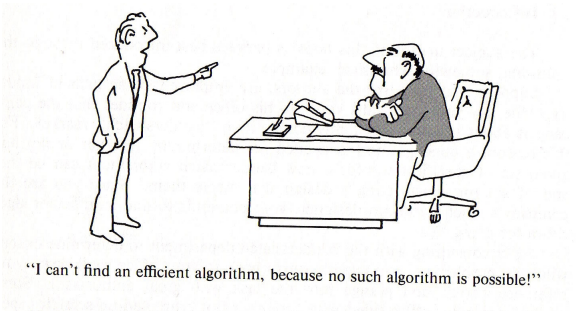


FIGURE 32 – Situation moins critiquable (figure repris de [6])

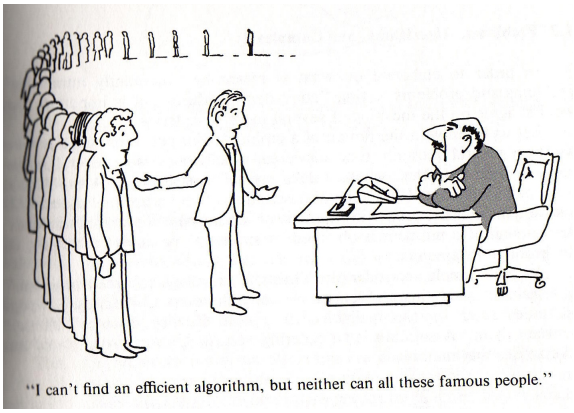


FIGURE 33 – Situation « scientifiquement » non critiquable (figure repris de [6])

5 Complexité Amortie

5.1 Introduction

Principe de la complexité amortie

— Amortissement = outil dans la conception et l'analyse des Structures de Données

— Implémentation avec des limites amorties (bien choisies) : Souvent plus simple et plus efficace que des implémentations dans le pire des cas

5.2 Illustration

5.2.1 Exemple de la gestion d'une Pile

Approche intuitive par l'exemple

Exemple 52 : Supposons la gestion d'une Pile d'éléments avec 3 primitives (Algorithme 5) :

— empiler un élément *el* dans une pile P,

— dépiler une pile P,

— et une primitive multi_pop qui effectue k dépilements.

Algorithme 5 : multi_pop(P,k)

Entrées : P : Pile, k : entier
Sorties : P : Pile

```
5.1 début
5.2     tant que ¬ pile_vide(P) et (k ≠ 0) faire
5.3         Dépiler(P);
5.4         k ← k − 1;
```

L'intérêt de calculer la complexité de multi_pop est d'envisager également que la pile a subi des opérations quelconques d'empilements et de dépilements.

Temps d'exécution des primitives sur une Pile P de s éléments

— empiler : $O(1)$

— dépiler : $O(1)$

— multi_pop : $\min(s, k)$

  — Si la pile contient $s > k$ éléments, k éléments dépilés de coût abstrait 1

  — sinon seulement s éléments dépilés de coût abstrait 1

Theorem 54. *Dans le pire des cas n opérations empiler, dépiler ou multi_pop suppose* $O(n^2)$.

Démonstration. En effet, l'opération multi_pop coûte $O(n)$, puisque la taille de la pile est n. Une séquence de n opérations coûte $O(n^2)$ pour multi_pop (Chaque opération coûte $O(n)$ dans une séquence de n opérations) Bien que cette analyse soit correcte, le résultat obtenu ne prend pas réellement en compte les comportements des primitives. □

5.2.2 Une idée simple : « Avoir une vision plus globale »

Soit une séquence d'opérations, calculer le temps global de la séquence et non des opérations individuelles

Exemple 53 : Pour n opérations, définir une limite souhaitée de la séquence $O(n)$ sans s'attacher à la complexité d'une opération.

5.3 Méthodes

5.3.1 Analyse Amortie

Technique d'Analyse Amortie

En général

- Prouver un résultat « le coût amorti cumulé est une limite sup. du coût actuel cumulé ».

$$\forall 1 \leq j \leq n, \sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} t_i \tag{22}$$

  avec :
  - a_i coût amorti de l'opération i
  - t_i coût actuel d'une opération i
  - n nombre total d'opérations
- « Gain accumulé » (*accumulated savings*) : Différence entre « coût accumulé amorti » et « coût actuel cumulé ».
- Amortissement autorise des opérations a avoir un coût qui dépasse leurs coûts amortis :
  - *expensive operations* : diminue le gain accumulé
  - *cheap operations* : augmente le gain accumulé

Idée

- Montrer que les « opérations coûteuses se produisent si seulement si le gain accumulé est suffisant pour absorber le coût restant ».
- Méthodes (équivalentes) proposées par Tarjan (1985) pour analyser les Structures de Données amorties :
  - méthode du banquier
    - Gains accumulés sont appelés des crédits
  - méthode du physicien
    - Gains accumulés sont appelés des potentiels

5.3.2 Méthode 1 : Méthode du banquier

Technique d'Analyse Amortie : méthode du banquier

Definition 55. Coût amorti a_i d'une opération est défini par :

$$a_i = t_i + c_i - \overline{c_i} \tag{23}$$

— t_i : coût actuel de l'opération

— c_i : crédits alloués par l'opération

— $\overline{c_i}$: crédits dépensés par l'opération

Règles principales :

— Chaque crédit doit être alloué avant d'être dépensé

— Un crédit est dépensé une seule fois (consommable)

— $\sum_i c_i \geq \sum_i \overline{c_i}$

Note : Définition d'un crédit invariant qui distribue les crédits (pour une opération de type *expensive*) : allocation de crédits pour ce coût.

Exemple 54 : Revenons au problème initial de la pile avec les trois primitives (empiler, dépiler et multi_pop).

Démonstration. Affectation des coûts amortis des différentes opérations (le crédit total ne doit jamais devenir négatif) (Tableau 12) :

TABLE 12 – Affectation des coûts pour la gestion d'une Pile

Opérations	coûts réels	coûts amortis
Empiler	1	2
Dépiler	1	0
Multi_pop	$\min(s, k)$	0

— Pour empiler, le coût amorti est de 2 (euros) et je consomme 1 euro pour l'opération ; il me reste donc un crédit de 1 euro.

— Si j'effectue une opération de dépilement (le coût amorti est de 0 mais je consomme 1 euro pris sur l'opération d'empilement (crédit est alors à zéro).

— (de même avec la primitive multi_pop).

Remarque :

— la primitive Multi_pop a un coût réel variable et un coût amorti nul !

— les coûts amortis des primitives sont en $O(1)$ (les coûts peuvent asymptotiquement être différents pour les différentes primitives)

— La pile contient toujours un nombre positif d'éléments, le crédit est toujours de valeur positive.

— pour une séquence quelconque de n opérations Empiler, Dépiler et Multi_Pop, le coût total amorti est une borne supérieure pour le coût total réel.

Conclusion : Comme le coût amorti total est $O(n)$, c'est aussi le coût total réel. □

5.3.3 Méthode du physicien

Technique d'Analyse Amortie : méthode du physicien

Definition 56. Soit la description d'une fonction définie par $\phi : d \to \Re^+$, d : objet (initialement à zéro) (Figure 34). Le Coût amorti a_i d'une opération est alors défini par :

$$a_i = t_i + \phi(d_i) - \phi(d_{i-1}) \tag{24}$$

avec :

— t_i : coût actuel de l'opération

— $\phi(d_i) - \phi(d_{i-1})$: différence de potentiel

FIGURE 34 – Représentation des opérations successives

Exercice 50 : Déterminer le coût actuel accumulé de la séquence d'opérations et conclure.

Démonstration.

$$\sum_{i=1}^{j} t_i = \sum_{i=1}^{j} (a_i + \phi(d_{i-1}) - \phi(d_i)) = \sum_{i=1}^{j} a_i + \phi(d_0) - \phi(d_j)$$

— Hypothèses : $\phi(d_0) = 0$ et $\phi(d_j) \geq \phi(d_0)$

— Conclusion : $\forall j, \sum_{i=1}^{j} a_i \geq \sum_{i=1}^{j} t_i$

i.e., les coûts amortis accumulés sont des limites sup. aux coûts actuels accumulés. □

Exemple 55 : Revenons au problème initial de la pile avec les trois primitives (empiler, dépiler et multi_pop). Déterminer le coût amorti.

Démonstration. Soit une fonction ϕ pour une structure de Données P_i (état de la Pile) :

$$\phi : P_i \rightarrow \text{ nombre d'éléments de la Pile}$$

Hypothèses vérifiées : $\phi(P_0) = 0$ (P_0 pile à l'état vide) ; $\phi(P_i) \geq 0$

Le coût amorti total par rapport à ϕ de n opérations représente donc une borne supérieure pour le coût réel. Calculons ainsi les coûts amortis des opérations de pile :

1. Si la i^{eme} opération sur une pile contenant s objets est une opération Empiler
   - $\phi(P_i) - \phi(P_{i-1}) = (s+1) - s = 1$
   - coût amorti : $a_i = t_i + \phi(d_i) - \phi(d_{i-1}) = 1 + 1 = 2$
2. Si la i^{eme} opération sur une pile est une opération Multi_pop ($k' = \min(k, s)$ objets dépilés)
   - $\phi(P_i) - \phi(P_{i-1}) = -k'$
   - coût amorti : $a_i = t_i + \phi(d_i) - \phi(d_{i-1}) = k' - k' = 0$ (idem pour dépiler)

Finalement, nous pouvons déterminer que :

- le coût amorti de chaque opération : $O(1)$
- le coût amorti total pour n opérations : $O(n)$
- Conclusion : Comme le coût amorti total est $O(n)$, c'est aussi le coût total réel.

□

5.3.4 Un exemple plus difficile

Un exemple de une gestion de liste avec maintien d'un invariant

Exemple 56 : Supposons une Structure de Données (notée SD) définie par un couple $\langle a, b \rangle$ avec a la tête de la liste, et b la fin de la liste dans l'ordre inversée. Soit une liste $[1, 2, 3, 4, 5, 6]$ avec une représentation possible de la forme :

- $a = [1, 2, 3]$ et $b = [6, 5, 4]$, avec :
- 1^{er} élément de a : tête de la liste complète,
- 1^{er} élément de b : dernier élément de la liste complète

Par hypothèse sur la gestion de la liste, nous essayons de maintenir l'invariant « a est vide si et seulement si b est vide » (la liste entière est vide). Nous pouvons ainsi conclure :

- si a vide et b non vide alors transfert le dernier élément de b dans a (migration d'une liste vers l'autre)
- En maintenant l'invariant, l'accès à la tête de liste est $O(1)$

Démonstration. Proposons les algorithmes de gestion de la liste, et déterminons leur complexité :

Fonction Tête($\langle a, b\rangle$) : entier

```
    Entrées : ⟨a, b⟩ : SD
-.1 début
-.2     si vide(a) alors
-.3         Tête ← nil
-.4     sinon
-.5         Tête ← 1^er élément de lst
```

Complexité Tête : $O(1)$

Algorithme 6 : Verif($\langle a, b\rangle$)

```
    Entrées : ⟨a, b⟩ : SD
    Sorties : ⟨a, b⟩ : SD
6.1 début
6.2     si vide(a) alors
6.3         a ← inverser(b);
6.4         b ← nil
```

Complexité Verif : $O(n)$

Algorithme 7 : Ajout($\langle a, b\rangle$, x)

```
    Entrées : ⟨a, b⟩ : SD, x : entier
    Sorties : ⟨a, b⟩ : SD
7.1 début
7.2     Verif(⟨a, b⟩);
7.3     Ajouter élément x dans b
```

Complexité Ajout : $O(n)$

Algorithme 8 : Gestion($\langle a, b\rangle$)

```
    Entrées : ⟨a, b⟩ : SD
    Sorties : ⟨a, b⟩ : SD
8.1 début
8.2     si vide(a) ou vide(b) alors
8.3         Verif(⟨b, a⟩);
8.4         x ← Tête(b);
8.5         b ← b privé de x;
8.6         Ajout(⟨a, b⟩ , x);
```

Complexité Gestion : $O(n)$

Illustrons ensuite le fonctionnement de l'algorithme *Gestion* (Tableau 13) :

TABLE 13 – Illustration fonctionnement de *Gestion*

Gestion([1,2,3],[6,5,4])	Gestion (nil,[6,5,4])	Gestion ([1,2,3],nil)
a=[1,2,3],	a = [4,5],	a = [1,2]
b = [6,5,4]	b = [6]	b = [3]

Utilisation la méthode du banquier Maintien d'un crédit invariant : chaque élément de la liste b est attribué un unique crédit (Tableau 14).

TABLE 14 – Méthode du banquier sur la gestion de liste avec maintien de l'invariant

Appel Algo.	coût actuel	crédits alloués	crédits dépensés	coût amorti
Ajout liste non vide	1 opér.	1	0	2
Gestion				
sans inversion	1 opér.	0	0	1
avec inversion	$m+1$ opér.	0	m	$m+1-m=1$

avec :

— m longueur de liste

— nouvel élément de b pour *Ajout* : 1 crédit alloué

Utilisation la méthode du physicien Soit une fonction ϕ : longueur de la liste b dans $\langle a, b\rangle$ (Tableau 15).

TABLE 15 – Méthode du physicien sur la gestion de liste avec maintien de l'invariant

Appel Algo.	coût actuel	Différence de Potentiel	coût amorti
Ajout liste non vide	1 opér.	1	2
Gestion			
sans inversion	1 opér.	0	1
avec inversion	m+1 opér.	- m	m+1-m=1

avec :

— m longueur de liste

— nouvel élément de b pour Ajout : 1 de différence de potentiel

— différence de potentiel pour gestion : 0 si pas d'inversion de liste (potentiel inchangé), $-m$ si inversion de liste (liste a à nil)

□

Exercice 51 : Nous savons que la recherche dichotomique effectue une recherche d'un élément de manière efficace à la condition que les éléments sont évidemment triés.

1. Rappeler la complexité temporelle d'une recherche dichotomique
2. Avant d'effectuer une recherche, cette liste doit être triée (par exemple, considérez un tri par insertion ou un tri par fusion). Déterminer la complexité globale d'un tri puis d'une recherche.
3. Supposons maintenant que pour n opérations, chaque élément est inséré par un tri puis on effectue une recherche dichotomique. Calculer la complexité amortie dans le pire des cas d'une séquence de n opérations de tri et de recherche d'éléments.

5.4 Conclusion

Conclusion

— La méthode du physicien peut paraître plus simple (dans la méthode du banquier, une difficulté réside dans le choix de la fonction qui alloue ou dépense des crédits)

— Des questions classiques : (i) Pourquoi allouer un crédit et dépenser aucun dans *Ajout* ? (ii) Pourquoi ne pas envisager une allocation de 2 et une dépense de 1 ? (la réponse est dans le choix de la fonction ϕ qui impose déjà un choix de décision)

6 Concl. Gen.

Conclusion Générale

Différentes Notions vues en cours

1. Plusieurs exemples illustrant de l'*intérêt des problèmes de complexité*
2. Algorithme efficace, ordre de grandeur asymptotique, Complexité temporelle vs Complexité espace mémoire, Efficacité locale (cas défavorable, cas moyen, cas favorable)
3. Algorithme Diviser pour Régner (relation de récurrences) : différentes méthodes de calcul de complexité (en particulier la méthode générale)
4. Différentes familles de complexité (problèmes **P**,**NP**, **NP – complet**)
5. Efficacité globale (Complexité amortie pour les Structures de Données)

Dernier Conseil

"Restez vigilant en terme de complexité sur les algo. à implémenter."

Conséquence de l'analyse sur le choix de la Structure de Données et des opérations en fonction de l'application envisagée

7 Bibliographie

Références

[1] A. Arnold et I. Guessarian *Mathématiques pour l'Informatique* Masson Eds., 2ème édition, 1997.

[2] T.H. Cormen, C.E. Leiserson, R. Rivest et C. Stein *Introduction à l'algorithmique* Dunod Eds., 2ème édition, 1994.

[3] J.P. Delahaye *L'intelligence et le calcul : de Gödel aux ordinateurs quantiques* Belin Pour la Science Eds, 2002.

[4] E. Fourrey *Récréations arithmétiques* Vuibert Eds, 2001.

[5] C. Froidevaux, M-Cl. Gaudel et M. Soria *Types de Données et Algorithmes* Mc Graw Hill, 1990.

[6] M.R. Garey et D.S. Johnson *Computer and Intractability : A guide to theory of NP-completeness* Freeman and Company Eds., New-York, 1979.

[7] P. Van Roy et S. Haridi *Programmation : Concepts, Techniques et Modèles* Dunod, 2007.

[8] C. Okasaki *Purely Functional Data Structures* Cambridge University Press, 1998.

[9] V.-Th. Paschos *Complexité et approximation polynomiale* Hermes Lavoisier, Paris, 2004.

[10] O. Ridoux et G. Lesventes *Calculateurs, Calculs, Calculabilité* Dunod, Paris, 2008.

[11] P. Wolper *Introduction à la calculabilité* Dunod, Paris, 3eme édition, 2006.

[12] M. Agrawal, N. Kayal et LN. Saxena PRIMES is in P. *Annals of Mathematics*, 160(2) : 781–793, 2002.

[13] M. Akra et L. Bazzi On the solution of linear recurrences equations. *Computational Optimization and Applications*, 10(2) :195–210, 1998.

[14] J. Bentley, D. Haken et J. Saxe A general method for solving divide and conquer recurrences. *SIGACT News*, 12(3) :36–44, 1980.

[15] S.A. Cook The complexity of Theorem Proving Procedures. *Proceedings of the third annual symposium on Theory of Computing*, May 1971 :151–158, 1971.

[16] B. Felgenhauer et F. Jarvis Enumerating possible Sudoku grids. 1–7, 2005.

[17] Tibor Rado On Non-Computable Functions. *Bell Systems Technology Journal*, 41(3) :877–884, May 1962.

A Annexe

A.1 Mesure de Performance

La mesure de performance dépend du matériel utilisé (processeur) et du système d'exploitation (Windows, UNIX, etc.). Il est donc conseillé de préci-

ser les caractéristiques de votre machine (e.g., puissance du processeur, espace mémoire) ainsi que le langage utilisé.

A.1.1 Temps CPU vs. Temps Utilisateur

Ne confondez pas le temps CPU avec le temps Utilisateur (en général, le temps utilisateur est plus élevé que le temps CPU) :

— Le temps CPU mesure uniquement le temps d'exécution d'un programme sur un processeur.

— Le temps utilisateur mesure le temps d'exécution du programme pour l'utilisateur. Mais ce temps prend également en compte les E/S, les autres tâches exécutées (système d'exploitation, par exemple), etc.

La mesure de performance s'intéresse au temps CPU et non au temps utilisateur. Pour des mesures de temps CPU, il est donc judicieux de ne pas surfer sur internet (augmentation possible de plus de 10% sur le temps utilisateur) et d'éviter (dans la mesure du possible) les instructions d'entrée/sortie (coûteuses également en temps liés à l'affichage et/ou à l'attente de saisie d'information).

De plus, pour chaque instance, il est important de calculer la moyenne sur plusieurs instances (avec les mêmes paramètres d'entrée). On considère qu'un échantillon représentatif minimum est de 20 instances. L'écart-type peut être aussi significatif pour votre analyse des résultats.

les instructions de mesure du temps dépendent du langage utilisé et du système d'exploitation.

A.1.2 Un exemple

Le programme suivant a été développé sous Windows avec l'environnement CodeBlocks [4].

Exemple 57 :

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <iomanip>
4
5  #include<time.h>
6
7  using namespace std;
8
9  #define MAX 1000001
10 #define SWAP(x,y,t)((t) = (x), (x) = (y), (y) = (t))
11
12 void tri1( int list[], int n)
13 {
14     int i, j, min, temp;
15     for (i=0; i < n-1; i++)
```

4. http ://www.codeblocks.org/ (accessible en 2014).

```
16      {
17          min = i;
18          for (j = i + 1; j <n; j++)
19              if (list[j] < list[min]) min = j;
20          SWAP (list[i], list[min], temp);
21      }
22  }
23
24  void fusionner ( int A [], int p,  int q, int r)
25  {
26       int n1;
27       int n2;
28      n1 = q - p + 1;
29      n2 = r - q;
30        int L [n1];
31        int R [n2];
32
33      for ( int i = 0; i < n1; i++) L[i]=A[p+i];
34      for ( int j = 0; j < n2; j++) R[j]= A[q+j+1];
35
36      L[n1]= MAX + 1;
37      R[n2] = MAX + 1;
38       int i=0, j = 0;
39      for ( int k= p; k <= r; k++)
40      {
41          if (L[i] <= R[j])
42          {
43              A[k] = L[i];
44              i = i + 1;
45          }
46          else
47          {
48              A[k] = R[j];
49              j = j + 1;
50          }
51      }
52  }
53
54  void tri_fusion( int A[],  int p,  int r)
55  {  int q;
56
57      if (p < r)
58      {
59          q =( p + r ) / 2;
60          tri_fusion (A, p, q);
61          tri_fusion (A, q+1, r);
62
63
64          fusionner (A, p, q, r);
65      }
```

```
66 }
67
68
69
70 void Fusion (int A[], int n)
71 {
72     tri_fusion (A, 0, n-1);
73
74 }
75
76 void tri_insertion (int A [], int n)
77 { int i, j, cle;
78
79
80     for (j = 1 ; j < n; j++)
81     {
82           cle = A[j];
83           i = j - 1;
84           while ((i >= 0) && (A[i] > cle))
85           {
86                 A[i+1] = A[i];
87                 i = i - 1;
88           }
89           A[i+1] = cle;
90     }
91
92 }
93
94 void Affich( int A[], int n)
95 {
96     int i;
97     for (i = 0; i < n; i++) cout<<"A["<<i<<"]="<<A[i]<<"␣";
98     cout<<endl;
99 }
100
101 double Mesure_temps (void (*fonction) (int B[], int c), int A[], int n)
102
103 {   time_t start, stop;
104     clock_t deb, fin;
105     double duree;
106
107
108 // methode 1
109 //              start = time(NULL);
110 //              tri_insertion(list1, val_max);
111 //              stop = time(NULL);
112 //              duree = ((double) difftime(stop,start));
113
114 // methode 2
115         deb = clock();
```

```
116           (*fonction) (A,n);
117           fin = clock();
118                   // CLK_TCK  : nombre de cycles d'horloges par seconde
119                   // CLOCKS_PER_SEC (ANSI C)
120           duree = ((double) (fin - deb)) / CLOCKS_PER_SEC;
121
122           return duree;
123   }
124
125   int* init (int val, int ch)
126   { int j;
127     int *tab;
128     if (val <= 0) return tab;
129
130     tab = new int [val];
131     if (tab == NULL)
132       {cout<<"probleme memoire\n";
133       system("PAUSE");
134       exit(1);
135       }
136     else
137     {
138       for (j = 0; j < val; j++)
139       {
140           if (ch == 1)    tab[j] = j;                       //cas favorable
141           else            tab[j] = val - j -1;      // generation cas defavorable
142       }
143     }
144      return tab;
145   }
146
147   int menu (int lg)
148   { // affichage des informations ...
149
150        int choix;
151       cout<<"Cas favorable : 1"<< endl;
152       cout<<"Cas defavorable : 2"<<endl;
153       cout<<"sinon les deux cas "<<endl;
154       cout<<" choix:";
155       cin>>choix;
156       if ((choix != 1) && (choix != 2)) choix = 3;
157
158       cout<<flush;
159       cout<<endl;
160       cout<<setw(lg)<<" iter no"<<"|"<<setw(lg)<< "Taille"<<"|";
161
162
163       cout<<setw(2*lg+1)<<"Insertion"<<"|";
164       cout<<setw(2*lg+1)<<"Fusion"<<"|"<<endl;
165
```

```
166       cout<<setw(lg)<<" "<<"|"<<setw(lg)<<" "<<"|";
167
168       cout<<setw(lg)<<"Favorable"<<"|"<<setw(lg)<<"Defavorable"<<"|";
169       cout<<setw(lg)<<"Favorable"<<"|"<<setw(lg)<<"Defavorable"<<"|";
170
171       cout<<endl;
172
173       return choix;
174   }
175
176   int main(int argc, char *argv[])
177   {
178       double duree;
179         int *list;
180
181       int nb_iter, iter, pas, val_min, val_max, lg;
182
183       cout<<"nb d'iterations:";
184       cin>> iter;
185       if (iter <= 0) iter = 1;
186
187       cout<<"taille min:";
188       cin>> val_min;
189       if (val_min < 0) val_min = 1;
190
191       if (iter >= 2)
192       {
193           cout<<"pas:";
194           cin>> pas;
195       }
196       else pas= 0;
197
198
199        val_max = val_min;              // taille initiale des tableaux
200
201       lg = 11;     // strlen("Defavorable");
202                    // longueur chaine la plus longue
203
204       nb_iter = 0;
205
206       int choix = menu(lg);
207
208
209     do
210       {
211                nb_iter = nb_iter + 1;
212                cout<<setw(lg)<<nb_iter<<"|"<<setw(lg)<< val_max<<"|";
213
214
215   // Tri par insertion
```

```
216             if ((choix ==1) || (choix == 3))
217             {
218                 list = init (val_max, 1);
219                 duree = Mesure_temps(tri_insertion, list, val_max);
220                 delete [] list;
221                 cout<<setw(lg)<<duree<<"|";
222             }
223             else cout<<setw(lg)<<"␣"<<"|";
224 
225             if ((choix == 2) || (choix == 3))
226             {
227                 list = init (val_max, 2);
228                 duree = Mesure_temps(tri_insertion, list, val_max);
229                 delete [] list;
230                 cout<<setw(lg)<<duree<<"|";
231             }
232             else cout<<setw(lg)<<"␣"<<"|";
233 
234 
235 // Fusion
236             if ((choix ==1) || (choix == 3))
237             {
238                 list = init (val_max, 1);
239                 duree = Mesure_temps(Fusion, list, val_max);
240                 delete [] list;
241                 cout<<setw(lg)<<duree<<"|";
242             }
243             else cout<<setw(lg)<<"␣"<<"|";
244 
245             if ((choix == 2) || (choix == 3))
246             {
247                 list = init (val_max, 2);
248                 duree = Mesure_temps(Fusion, list, val_max);
249                 delete [] list;
250                 cout<<setw(lg)<<duree<<"|";
251             }
252             else cout<<setw(lg)<<"␣"<<"|";
253 
254 
255     cout<<endl;
256 
257     if (nb_iter < iter)      val_max = val_max + pas;
258 
259     }
260     while  (nb_iter< iter);
261 
262     system("PAUSE");
263     return EXIT_SUCCESS;
264 }
```

A.2 Compléments Math

A.2.1 Rappels Math "classiques"

1. Monotonie
   - Une fonction $f(n)$ est monotone croissante si $m \leq n$ implique $f(m) \leq f(n)$
   - Une fonction $f(n)$ est monotone décroissante si $m \leq n$ implique $f(m) \geq f(n)$
   - Une fonction $f(n)$ est strictement croissante si $m < n$ implique $f(m) < f(n)$
   - Une fonction $f(n)$ est strictement décroissante si $m < n$ implique $f(m) > f(n)$
2. Parties entières $(n \geq 0, a > 0, b > 0)$
   - $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
   - $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n$
   - $\left\lceil \frac{\lceil \frac{n}{a} \rceil}{b} \right\rceil = \lceil \frac{n}{a.b} \rceil$
   - $\left\lfloor \frac{\lfloor \frac{n}{a} \rfloor}{b} \right\rfloor = \lfloor \frac{n}{a.b} \rfloor$
   - $\lceil \frac{a}{b} \rceil \leq \frac{a+(b-1)}{b}$
   - $\lfloor \frac{a}{b} \rfloor \geq \frac{a-(b-1)}{b}$
3. Congruences
   - $a \ mod \ n = a - \lfloor \frac{a}{n} \rfloor .n$
   - si $a \ mod \ n = b \ mod \ n$ alors $a \equiv b \ (mod \ n)$ ("a congru à b modulo n")
4. Sommation
   - $\sum_{i=0}^{n} i = \frac{n.(n+1)}{2}$
   - $\sum_{i=2}^{n} i = \frac{n.(n+1)}{2} - 1$
   - $\sum_{i=2}^{n} (i-1) = \frac{n.(n-1)}{2}$
   - $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$
   - $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$, pour $|x| < 1$
5. Puissances a^n
   - $a^0 = 1$
   - $a^1 = a$
   - $a^{-1} = \frac{1}{a}$
   - $((a^m)^n) = a^{m.n}$
   - $((a^m)^n) = ((a^n)^m)$
   - $a^m.a^n = a^{m+n}$

- La fonction a^n est monotone croissante pour $a \geq 1, \forall n$
- Toute fonction dont la base $a > 1$ croit plus vite que toute fonction polynomiale, i.e.
  - a, b constantes réelles, $a > 1$, $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$
  - d'où $n^b = o(a^n)$

6. Exponentielles e^x
   - $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$
   - $\forall x$ réel, $e^x \geq 1 + x$; d'où : $x \to 0, e^x = 1 + x + \theta(x^2)$
   - $|x| \geq 1,\ 1 + x \leq e^x \leq 1 + x + x^2$
   - $\lim_{n\to\infty}(1 + \frac{x}{n})^n = e^x$
7. Logarithmes $\log_b(n)$ $(a, b, c > 0$ réels$)$
   - $a = b^{\log_b(a)}$
   - $\log_c(a.b) = \log_c(a) + \log_c(b)$
   - $\log_b(a^n) = n.\log_b(a)$
   - $\log_b(a) = \frac{\log_c(a)}{\log_c(b)}$
   - $\log_b(\frac{1}{a}) = -\log_b(a)$
   - $\log_b(a) = \frac{1}{\log_a(b)}$
   - $a^{\log_b(c)} = c^{\log_b(a)}$
   - La fonction $\log_b n$ est monotone croissante pour $b > 1,\ \forall n > 0$
8. Logarithme népérien $\ln x$
   - $\ln(1 + x) = \log_e(1 + x)$
   - $\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots,\ |x| < 1$
   - $\frac{x}{1+x} \leq \ln(1 + x) \leq x, x \geq -1$
9. Factorielle $n!$
   - $n! \leq n^n$
   - $n! = \sqrt{2.\pi.n}.(\frac{n}{e})^n.(1 + \Theta(\frac{1}{n}))$ (formule de Stirling)
   - $n! = \sqrt{2.\pi.n}.(\frac{n}{e})^n.e^{\alpha_n}$ avec $\frac{1}{12.n+1} < \alpha_n < \frac{1}{12.n}$

Exercice 52 : Vérifier - $n! = o(n^n)$, $n! = \omega(2^n)$ et $\log_2(n!) = \Theta(n.\log_2(n))$

10. Itération fonctionnelle

$$f^{(i)}(n) = \begin{cases} n & \text{si } i = 0, \\ f^{(i)}(f^{(i-1)}(n)) & \text{sinon} \end{cases}$$

Exercice 53 : Montrer que $f(n) = 2.n.f^{(i)}(n)$ permet d'obtenir $f^{(i)}(n) = 2^i.n$

11. Logarithme itéré

$$\log_2^*(n) = \min\left\{i \geq 0 : \log_2^{(i)}(n) \leq 1\right\}$$

Exercice 54 : Montrer que nous obtenons les valeurs suivantes pour les logarithmes itérés $\log_2^*(2) = 1$, $\log_2^*(4) = 2$, $\log_2^*(16) = 3$, $\log_2^*(65536) = 4$ et $\log_2^*(2^{65536}) = 5$

12. Nombre de Fibonacci F_i
    - $\phi = \frac{1+\sqrt{5}}{2}$, $\overline{\phi} = \frac{1-\sqrt{5}}{2}$, $F_i = \frac{\phi - \overline{\phi}}{\sqrt{5}}$

Exercice 55 : Montrer que le nombre de Fibonacci augmente de façon exponentielle, i.e. $F_i \approx \frac{\phi^i}{\sqrt{5}}$

A.2.2 Notation asymptotique : compléments

1. Comparaison de fonctions asymptotiquement positives $f(n)$ et $g(n)$
    - Réflexivité
        - $f(n) = \Theta(f(n))$
        - $f(n) = O(f(n))$
        - $f(n) = \Omega(f(n))$
    - Symétrie
        - $f(n) = \Theta(g(n))$ si et seulement si $g(n) = \Theta(f(n))$
    - Symétrie transposée
        - $f(n) = O(g(n))$ si et seulement si $g(n) = \Omega(f(n))$
        - $f(n) = o(g(n))$ si et seulement si $g(n) = \omega(f(n))$
    - Transitivité
        - $f(n) = \Theta(g(n))$ et $g(n) = \Theta(h(n))$ implique $f(n) = \Theta(h(n))$
        - $f(n) = O(g(n))$ et $g(n) = O(h(n))$ implique $f(n) = O(h(n))$
        - $f(n) = \Omega(g(n))$ et $g(n) = \Omega(h(n))$ implique $f(n) = \Omega(h(n))$
        - $f(n) = o(g(n))$ et $g(n) = o(h(n))$ implique $f(n) = o(h(n))$
        - $f(n) = \omega(g(n))$ et $g(n) = \omega(h(n))$ implique $f(n) = \omega(h(n))$
2. Analogie avec les réels

$f(n), g(n)$ asymptotiquement positifs	a,b Réels
$f(n) = O(g(n))$	$a \leq b$
$f(n) = \Omega(g(n))$	$a \geq b$
$f(n) = \Theta(g(n))$	$a = b$
$f(n) = o(g(n))$	$a < b$
$f(n) = \omega(g(n))$	$a > b$

— $f(n) = o(g(n))$: $f(n)$ est asymptotiquement inférieure à $g(n)$
— $f(n) = \omega(g(n))$: $f(n)$ est asymptotiquement supérieure à $g(n)$
— Attention : deux fonctions ne sont pas toujours comparables asymptotiquement (deux réels peuvent être comparés), i.e. impossible d'avoir : $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$.

Exemple 58 : n et $n^{1+\sin(x)}$ fonctions non comparables à l'aide d'une notation asymptotique

3. Polynômes $p(n) = \sum_{i=0}^{d} a_i.n^i$, a_i constantes et $a_d \neq 0$
   — est asymptotiquement positif si et seulement si $a_d > 0 : p(n) = \Theta(n^d)$;
   — est une fonction monotone croissante pour $a_d > 0$;
   — est une fonction monotone décroissante pour $a_d < 0$;
   — On dit que $f(n)$ a une borne polynomiale si $f(n) = \Theta(n^k)$, k donné.
4. Logarithmes
   — On dit que $f(n)$ a une borne polylogarithmique si $f(n) = O(\log_2^k(n))$, k constante
   — Toute fonction polynomiale croît plus vite que toute fonction polylogarithme

Exercice 56 : Montrer la relation entre les polynômes et les polylogarithmes

Nous savons que $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$ et en substituant n par $\log_2(n)$ et en substituant a par 2^a. Nous obtenons : $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$

$\lim_{n\to\infty} \frac{\log_2^b(n)}{(2^a)^{\log_2(n)}} = 0$, i.e. $\lim_{n\to\infty} \frac{\log_2^b(n)}{n^a} = 0$. Donc $\log_2^b(n) = o(n^a)$, a constante positive. cqfd

A.3 Algorithmes de Tri

A.3.1 Tri par insertion

Algorithme 9 : Tri_insertion(A,n)

Entrées : $A[1..n]$ non triée, n entier
Sorties : $A[1..n]$ triée

```
9.1 début
9.2   pour j ← 2 à n faire
9.3     cle ← A [j];
9.4     // insere A[j] dans la séquence triée A[1..j - 1]
9.5     i ← j − 1;
9.6     tant que (i > 0) et (A[i]> cle) faire
9.7       A[i + 1] ← A[i];
9.8       i ← i − 1;
9.9     A[i + 1] ← cle;
```

A.3.2 Tri par sélection

Algorithme 10 : Tri_selection(A,n)

```
        Entrées : A[1..n] non triée, n entier
        Sorties : A[1..n] triée
10.1    début
10.2        pour i ← 0 à n faire
10.3            min ← 0 ;
10.4            pour j ← i + 1 à n faire
10.5                si (A[j] < A[min]) alors
10.6                    min ← j ;
10.7            si (min ≠ j) alors
10.8                permuter(A[min],A[j])
```

A.3.3 Tri à bulles

Le tri à bulles classique :

Algorithme 11 : Tri_bulles(A,n)

```
        Entrées : A[1..n] non triée, n entier
        Sorties : A[1..n] triée
11.1    début
11.2        echange ← VRAI ;
11.3        tant que echange faire
11.4            echange ← FAUX ;
11.5            pour i ← 1 à n − 1 faire
11.6                si (A[i] > A[i + 1]) alors
11.7                    permuter(A[i],A[i + 1]) ;
11.8                    echange ← VRAI ;
```

Une version du tri à bulles : Tri cocktail

Algorithme 12 : Tri_cocktail(A,n)

```
     Entrées : A[1..n] non triée, n entier
     Sorties : A[1..n] triée
12.1 début
12.2     echange ← VRAI ;
12.3     deb ← 1 ;
12.4     fin ← n − 1 ;
12.5     tant que echange faire
12.6         echange ← FAUX ;
12.7         pour i ← deb à fin faire
12.8             si (A[i] > A[i + 1]) alors
12.9                 permuter(A[i],A[i + 1]) ;
12.10                echange ← VRAI ;
12.11        fin ← n − 1 ;
12.12        pour i ← fin à deb faire
12.13            si (A[i] > A[i + 1]) alors
12.14                permuter(A[i],A[i + 1]) ;
12.15                echange ← VRAI ;
12.16        deb ← deb + 1 ;
```

A.3.4 Tri rapide

Algorithme 13 : Tri_rapide(A,n)

```
     Entrées : A[1..n] non triée, n entier
     Sorties : A[1..n] triée
13.1 début
13.2     si p < r alors
13.3         q ← Partition(A,p,r) ;
13.4         Tri_rapide (A,p,q − 1) ;
13.5         Tri_rapide (A,q + 1, r) ;
```

Fonction Partition(A,p,r) : entier

```
-.1 début
-.2   x ← A[r] ;
-.3   i ← p − 1 ;
-.4   pour j ← p à r − 1 faire
-.5     si A[j] ≤ x alors
-.6       i ← i + 1 ;
-.7       permuter(A[i],A[j]) ;
-.8   permuter(A[i + 1],A[r]) ;
-.9   retourner i + 1 ;
```

A.3.5 Tri par tas

Fonction Parent(i) : entier

```
-.1 début
-.2   retourner ⌊i/2⌋ ;
```

Fonction Gauche(i) : entier

```
-.1 début
-.2   retourner 2 × i ;
```

Fonction Droite(i) : entier

```
-.1 début
-.2   retourner 2 × i + 1 ;
```

Algorithme 14 : Entasser_Max(A,i,nb)

```
14.1 début
14.2   left ← Gauche(i) ;
14.3   right ← Droite(i) ;
14.4   si left ≤ nb et A[left] > A[i] alors
14.5     max ← left ;
14.6   sinon
14.7     max ← i ;
14.8   si right ≤ nb et A[right] > A[max] alors
14.9     max ← right ;
14.10  si max ≠ i alors
14.11    permuter(A[i],A[max]);
14.12    Entasser_Max(A,max,nb) ;
```

Algorithme 15 : Construire_Tas_Max(A,n)

```
15.1 début
15.2     pour i ← ⌊n/2⌋ à 1 faire
15.3         Entasser_Max(A,i,n) ;
```

Algorithme 16 : Tri_par_Tas(A,n)

```
16.1 début
16.2     Construire_Tas_Max(A,n) ;
16.3     nb ← n ;
16.4     pour i ← n à 2 faire
16.5         permuter(A[1],A[i]);
16.6         nb ← nb − 1 ;
16.7         Entasser_Max(A,1,nb) ;
```

A.4 fiches de TD

A.4.1 TD no 1

fiche de TD 1

Exercice 1 : **Tri par sélection.** On considère le tri suivant de n nombres rangés dans un tableau A : on commence par trouver le plus petit élément de A et on le permute avec $A[1]$. On continue de cette manière pour les $n-1$ premiers éléments de A. Écrire le pseudo-code pour cet algorithme. Donner le temps d'exécution pour le cas optimal et le pire des cas.

Exercice 2 : Le **Tri à bulles** est un algorithme défini par (Cf. annexe pour une version optimisée) :

Algorithme 17 : Tri_Bulles(A,n)

```
      Entrées : A : tableau, n : entier
17.1  début
17.2      pour i ← 1 à n faire
17.3          pour j ← n à i + 1 faire
17.4              si (A[j] < A[j − 1]) alors
17.5                  permuter(A[j], A[j − 1]) ;
```

1. Quel est le temps d'exécution du tri à bulles dans le cas le plus défavorable ?
2. Quelles sont ses performances, comparées au tri par insertion et au tri fusion ?
3. Quel est le temps d'exécution du tri cocktail dans le pire des cas (optionnel) ?

Exercice 3 : Le tri par insertion peut être exprimé sous la forme d'une procédure récursive de la manière suivante. Pour trier $A[1..n]$, on trie récursivement $A[1..n-1]$ puis on insère $A[n]$ dans le tableau trié $A[1..n-1]$. Écrire une récurrence pour le temps d'exécution de cette version récursive du tri par insertion.

Exercice 4 : Déterminer la complexité dans le cas favorable et le cas défavorable pour la recherche d'un palindrome.

Exercice 5 : **Recherche séquentielle** : Déterminer les complexités (cas favorable, cas moyen, cas défavorable) pour l'algorithme de recherche séquentielle d'un élément dans une liste.

A.4.2 TD no 2

fiche de TD 2

Exercice 1 : Bien que dans le pire des cas, le tri par fusion s'exécute en $\Theta(n.\log_2(n))$ et le tri par insertion en $\Theta(n^2)$, les facteurs constants du tri par insertion le rendent plus rapide pour n petit. Il est naturel d'utiliser le tri par insertion à l'intérieur du tri par fusion lorsque les sous-problèmes deviennent suffisamment petits. On va étudier la modification suivante du tri par fusion : $\frac{n}{k}$ sous-listes de longueur k sont triées via le tri par insertion, puis fusionnées à l'aide du mécanisme de fusion classique, k est une valeur à déterminer.

1. Montrer que les $\frac{n}{k}$ sous-listes, chacune de longueur k, peuvent être triées via le tri par insertion avec un temps $\Theta(n.k)$ dans le cas le plus défavorable.
2. Montrer que les sous-listes peuvent être fusionnées en n : $\Theta(n.\log_2(\frac{n}{k}))$ dans le cas le plus défavorable.
3. Sachant que l'algorithme modifié s'exécute en n : $\Theta(n.k+n.\log_2(\frac{n}{k}))$ dans le cas le plus défavorable, quelle est la plus grande valeur asymptotique (notation Θ) de k en tant que fonction de n, pour lequel l'algorithme modifié a le même temps d'exécution asymptotique que le tri par fusion classique.
4. Comment doit-on choisir k en pratique ?

Exercice 2 : Déterminer la complexité temps de la procédure de Hanoï (inventé par Edouard Lucas, 1892)

Procédure Hanoi(n,A,B,C)

```
    Entrées : n : entier, A,B,C : caractères
-.1 début
-.2 |  // A : départ, B : intermédiaire, C : Arrivée ;
-.3 |  si (N = 1) alors
-.4 |  |  écrire(" Transférer de A vers C ")
-.5 |  sinon
-.6 |  |  Hanoi (n − 1, A, C,B) ;
-.7 |  |  Ecrire (" Transferer de A vers C ") ;
-.8 |  |  Hanoi (n − 1,B,A,C);
```

Exercice 3 : **Recherche Dichotomique et variante** En supposant une liste A triée d'éléments, on peut comparer le milieu de la séquence avec la valeur v recherchée, et supprimer la moitié de la séquence de la suite des opérations. La recherche dichotomique est un algorithme qui répète cette procédure, en divisant par deux à chaque fois la taille de la partie restante de la liste.

1. Écrire le pseudo code (itératif ou récursif) de la recherche dichotomique. Expliquez pourquoi le temps d'exécution de la recherche dichotomique dans le cas le plus défavorable est $\Theta(\log_2(n))$? Question facultative : temps d'exécution de la recherche dans le cas favorable ?

2. Déterminer la complexité temporelle pour la recherche trichotomique (algorithme identique à la recherche dichotomique mais avec une décomposition en 3 partitions correspondant à trois sous-listes).
3. Effectuer la même étude mais en supposant k partitions ?
4. On souhaite une performance de la recherche " k-tomique " au moins c fois plus rapide que pour une recherche dichotomique simple. Déterminer la valeur de k pour des valeurs de $c = 2$ et $c = 10$?

A.4.3 TD no 3

fiche de TD 3

Exercice 1 : Quelles sont les conséquences de trois stratégies de passage de paramètre d'un tableau de N éléments

- Un tableau est passé par adresse : le temps d'exécution est constant $T(N) = \Theta(1)$
- Un tableau est passé par copie : le temps d'exécution est linéaire $T(N) = \Theta(N)$
- Un tableau est passé via une copie uniquement du sous-intervalle susceptible d'être utilisé par la procédure appelé : temps linéaire $T(N) = \Theta(q - p + 1)$ si le sous-tableau $A[p..q]$ est transmis

1. On considère l'algorithme récursif de recherche dichotomique consistant à trouver un nombre dans un tableau trié. Donner les récurrences correspondant aux temps d'exécution, pour chacune des stratégies proposées précédemment de passage de tableau, et donner des bonnes bornes supérieures pour les solutions de récurrences. N est la taille du problème initial et n la taille d'un sous-problème.
2. Même question pour l'algorithme Tri-fusion
3. idem pour le tri par insertion

Exercice 2 : Nous souhaitons effectuer une comparaison entre deux algorithmes en terme de temps et d'espace mémoire : le premier est itératif, le second est récursif. Ces algorithmes ont pour objet la somme de n nombres réels déjà définis dans un tableau $A[1..n]$.

Exercice 3 : On propose une fonction recherchant l'élément maximum dans une liste non triée

1. En supposant le nombre de comparaisons comme opération fondamentale, déterminer les coûts pour les cas favorables et défavorables
2. En supposant le nombre d'affectations comme opération fondamentale, déterminer les coûts pour les cas favorables et défavorables
3. Déterminer le pire des cas pour la recherche du 2ème élément maximum du tableau
4. Pouvons nous trouver un algorithme qui serait plus rapide ? La réponse est évidemment " oui ", proposez le ! ! Déterminer son coût dans le pire des cas ? Avons-nous améliorer par la même occasion, la recherche du 2ème élément maximum de la liste ?

A.4.4 TD no 4

fiche de TD 4

Exercice 1 : Utiliser la méthode générale pour donner des bornes asymptotiques approchées pour les récurrences suivantes :

1. $T(n) = 2.T(\frac{n}{2}) + n$
2. $T(n) = 2.T(\frac{n}{2}) + 1$
3. $T(n) = 4.T(\frac{n}{2}) + n$
4. $T(n) = 4.T(\frac{n}{2}) + n^2$
5. $T(n) = 4.T(\frac{n}{2}) + n^3$

Exercice 2 : Le temps d'exécution d'un algorithme A est décrit par la récurrence $T(n) = 7.T(\frac{n}{2}) + n^2$. Un algorithme concurrent A' a un temps d'exécution décrit par $T'(n) = a'.T'(\frac{n}{4}) + n^2$

1. Appliquer la méthode générale et la méthode des arbres récursifs pour déterminer les fonctions asymptotiques correspondantes.
2. Quelle est la plus grande valeur entière de a' telle que A' soit asymptotiquement plus rapide que A ?

Exercice 3 : La méthode générale n'est pas applicable.

1. Donner une borne à la récurrence $T(n) = T(\lfloor \frac{n}{3} \rfloor) + T(\lfloor \frac{2.n}{3} \rfloor) + \Theta(n)$
2. La méthode générale est-elle applicable à la récurrence $T(n) = 4.T(\frac{n}{2}) + \Theta(n^2.\log_2(n))$? Pourquoi ? Donner une borne supérieure pour cette récurrence
3. Donner une borne supérieure pour la récurrence suivante $T(n) = 2.T(\frac{n}{2}) + \Theta(n.\log_2(n))$

A.4.5 TD no 5

fiche de TD 5

Exercice 1 : Décrire un algorithme en $\Theta(n.\log_2(n))$ qui étant donnés un ensemble S de n entiers et un autre entier x, détermine s'il existe ou non deux éléments de S dont la somme vaut exactement x.

Exercice 2 : Déterminer la complexité du pgcd de deux nombres n et m.

Exercice 3 : Trouver deux nombres entiers compris entre 1 et (N = 1000) tels, qu'étant multipliés entre eux ils donnent un certain produit, et que renversés ils donnent comme nouveau produit le premier renversé [4]. Par exemple :

- $41 \times 2 = 82$ et $14 \times 2 = 28$
- $32 \times 21 = 672$ et $23 \times 12 = 276$
- $312 \times 221 = 68952$ et $213 \times 122 = 25986$

Question supplémentaire : Déterminer le nombre de solutions satisfaisant cette propriété pour N quelconque.

Exercice 4 : Proposez en pseudo-langage un algorithme de recherche de toutes les permutations d'une chaîne de caractères. Calculez sa complexité.

Exercice 5 : Le **problème des n-reines** consiste à placer n reines sur un échiquier de $n \times n$ cases, sans qu'aucune reine n'en menace une autre. Pour $n = 8$, nous pouvons montrer que le nombre de solutions est de 92. Comme dans le jeu d'échecs, la reine menace une autre pièce de l'échiquier située sur la même ligne, colonne ou diagonale.

1. Pour un échiquier de taille $n \times n$, le nombre max. de reines, noté $R(n)$, est contraint par $R(n) \leqslant n$.
2. En Proposer une stratégie et déterminer les solutions pour $n = 4$ (optionnel de même pour les cas simples : $n = 2$ ou $n = 3$).
3. Décrire l'algorithme (algorithme de *backtracking*) :
   - L'algorithme *Libre* teste si la case à la ligne *lig* et à la colonne *col* n'est pas menacée par les reines déjà placées.
   - L'algorithme *Placer* est chargé de positionner les reines les unes après les autres en parcourant l'arbre de décision. L'algorithme doit donner toutes les solutions (i.e., poursuit sa recherche).
4. En déterminer la complexité de l'algorithme.

A.4.6 TD no 6

fiche de TD 6

Exercice 1 : La suite de Fibonacci est définie par 1, 1, 2, 3, 5 etc. On veut calculer le n-ième terme de la suite de manière efficace

1. Proposer un algorithme récursif. Donner le nombre d'appels récursifs de cet algorithme.
2. Proposer une variante où chaque valeur de la suite n'est calculée qu'une seule fois en utilisant de la mémoire.
3. Donner une solution où la complexité est logarithmique en n (optionnel)

Exercice 2 : Le problème est de déterminer à partir de quel étage d'un immeuble sauter par la fenêtre est fatal. Vous êtes dans un immeuble à n étages (numérotés de 1 à n) et vous disposez de k étudiants. Il n'y a qu'une seule opération possible pour tester si la hauteur d'un étage est fatale : faire sauter un étudiant par la fenêtre. S'il survit, vous pouvez le ré-utiliser ensuite, sinon, vous ne pouvez plus. Vous devez proposer un algorithme pour trouver la hauteur à partir de laquelle un saut est fatal (renvoyer $n+1$ si on survit encore au n étage) en faisant le minimum de sauts.

1. Si $k \geqslant \lceil \log_2(n) \rceil$, proposer un algorithme en $O(\log_2(n))$ sauts.
2. Si $k < \lceil \log_2(n) \rceil$, proposer un algorithme en $O\left(k + \frac{n}{2^{k-1}}\right)$ sauts.
3. Si $k = 2$, proposer un algorithme en $2.\sqrt{n}$ sauts.
4. Dans ce dernier cas, proposer aussi un algorithme en $\sqrt{2.n}$ sauts.

Exercice 3 : Le **problème k-somme** peut être défini par : Étant donnés un ensemble E de n entiers et un entier S, déterminer s'il existe $e_1, e_2, \ldots, e_k$ distincts dans E tels que $e_1 + e_2 + \ldots + e_k = S$.

1. Proposer un algorithme simple pour résoudre le problème 2-somme avec une complexité quadratique
2. En généralisant cette méthode, en déduire une première borne supérieure polynomiale sur la complexité du problème k-somme, pour $k \geq 2$.
3. Proposer un algorithme permettant de résoudre le problème 2-somme sur une liste triée en complexité linéaire.
4. En déduire un algorithme de complexité quadratique pour 3-somme.
5. Proposer un algorithme de complexité $O(n.\log_2(n))$ pour 2-somme avec une liste non triée.
6. Question difficile : Proposer un algorithme de complexité $O(n^2.\log_2(n))$ pour 4-somme (on pourra utiliser des tableaux auxiliaires).
7. Question difficile : En déduire un algorithme résolvant k-somme lorsque k est pair en complexité $O(n^{\frac{k}{2}}.\log_2(n))$, et une variante en $O(n^{\frac{k+1}{2}})$ lorsque k est impair.

Exercice 4 : **Mariage Stable** : Supposons n hommes et n femmes qui veulent se marier. Étant donné :

— Pour chaque homme, la liste comprenant les n femmes en ordre de préférence, et

— Pour chaque femme, la liste comprenant les n hommes en ordre de préférence

Le problème est de trouver un mariage entre les hommes et les femmes qui est **stable**. Un mariage est dit **stable** s'il n'y a aucune personne X telle que : X préfère une personne Y à son partenaire actuel, et en même temps, Y préfère la personne X à son partenaire actuel.

1. Déterminer le(s) mariage(s) stables pour les deux cas ci-dessous ($n = 2$) :
   (a) Soient les préférences suivantes concernant les quatre personnes (André, Bernard, Daniella et Maria) :

Personne	**choix** 1	**choix** 2
André	Daniella	Maria
Bernard	Daniella	Maria
Daniella	André	Bernard
Maria	Bernard	André

   (b) Reprendre le même énoncé avec *Bernard* qui préfère *Maria* à *Daniella*.
2. Proposer un algorithme qui détermine les différents couples suivant leur liste de préférences (liste de préférences complète pour chaque individu). En déterminer la complexité de l'algorithme.
3. Nous savons que certains individus préfèrent rester célibataires que d'être "mal-mariés" (une personne qui ne leur plait pas suffisamment). Proposer une solution permettant de modéliser une liste de préférences devenant incomplète.

A.5 fiches de TP

Consignes :

- vous êtes libre d'utiliser le langage de programmation de votre choix. Si nécessaire, vous devrez préciser les directives de compilation dans un fichier README. Le programme doit être facilement exécutable et compréhensible (pensez également à commenter votre programme pour améliorer sa lisibilité).
- Un compte-Rendu (au format pdf) devra obligatoirement contenir les sections suivantes :
  - une page de garde (noms/prénoms du groupe, date, formation),
  - une introduction,
  - une analyse de votre programme en pseudo-code (le code sera également fourni).
  - les résultats théoriques en terme de complexité (Cf. cours et/ou TD ; si le problème n'a pas été résolu, il vous appartiendra d'effectuer la démonstration correspondante)
  - les résultats numériques obtenus,
  - Une étude comparative entre les résultats théoriques et les résultats pratiques. N'oubliez pas également de préciser la configuration de votre ordinateur sur laquelle vous avez obtenu vos résultats.
  - une conclusion.

Certains exercices requièrent de tracer des courbes relatives aux temps d'exécution de vos programmes. Pour tracer vos graphiques, vous pouvez utiliser des outils tels que OpenOffice, GnuPlot, etc.

A.5.1 fiche de TP no 1

Objectif : L'objectif de ce TP est de vous familiariser avec les ordres de grandeur en complexité

Hanoi Écrivez un programme permettant de résoudre le problème des tours de Hanoï pour un entier n passé en paramètre.

- Effectuez plusieurs mesures du temps d'exécution ($n = 1, 2, 5, 10, 15, \ldots$) et tracez la courbe résultante ainsi qu'une courbe de tendance.
- Comparez vos résultats avec la complexité théorique calculée en TD.

Complexité et récursivité Écrivez le programme permettant de calculer la suite récurrente suivante :

$$u_n = \begin{cases} 1 & si \quad n = 0 \quad ou \quad n = 1 \\ u_{n-1} + u_{n-2} & si \quad n \geq 2 \end{cases}$$

— Proposer un algorithme itératif et Mesurer les temps d'exécution pour différentes valeurs de n.

— Proposer une version récursive et effectuer la même opération. Constatez-vous une amélioration des temps d'exécution ?

— Il existe un algorithme de complexité logarithmique pour calculer cette suite, expliquez le, implantez le et expérimentez.

Crible d'Eratosthène Eratosthène était un savant grec qui avait proposé une méthode de recherche des nombres premiers inférieurs à N. Le principe du crible est simple. On se donne un entier N arbitraire et on constitué initialement la liste des entiers compris entre 1 et N. L'algorithme consiste à *cribler* (cocher) tous les nombres qui ne sont pas premiers de la manière suivante :

1. On se place au début de la liste et on coche 1
2. On avance jusqu'au prochain nombre p qui n'a pas été coché, et on crible ses multiples
3. On recommence au point 2 tant qu'on n'a pas atteint la fin de la liste

Proposer un algorithme qui effectue cette tâche, et calculer sa complexité.

A.5.2 fiche de TP no 2

Objectif : Mesurer la complexité d'un algorithme d'exploration : "Le compte est bon"

Dans un célèbre jeu télévisé, les candidats doivent en 1 minute atteindre un résultat donné en utilisant 6 nombres et les quatre opérations arithmétiques : +, −, ×, /. Ce résultat est compris entre 100 et 999 et les nombres sont tirés au sort parmi un ensemble de 28 plaques composé de :

— 20 plaques numérotées de 1 à 10 (2 par nombre)

— 2 plaques de 25

— 2 plaques de 50

— 2 plaques de 75

— 2 plaques de 100

Les calculs doivent suivre les règles suivantes :

- Chaque nombre ne peut être utilisé qu'une seule fois.
- Les opérations sont restreintes aux entiers naturels positifs. Par exemple : les divisions ne sont autorisées que si le reste est nul. De même les soustractions renvoyant un résultat négatif sont interdites.

Ce jeu peut être résolu en utilisant la technique d'exploration suivante. Soit R le résultat à atteindre et P_n l'ensemble des n plaques tirées au sort. Considérons toutes les paires (p_i, p_j) de deux entiers pris dans P_n. Pour chaque (p_i, p_j) il est possible d'appliquer l'une des quatre opérations et construire ainsi 4 nouveaux ensembles :

$P_{n-1} = (p_i + p_j) \cup (P_n - \{p_i, p_j\})$
$P_{n-1} = (p_i \times p_j) \cup (P_n - \{p_i, p_j\})$
$P_{n-1} = (p_i - p_j) \cup (P_n - \{p_i, p_j\})$
$P_{n-1} = (p_i / p_j) \cup (P_n - \{p_i, p_j\})$

Le processus d'exploration consiste à tester successivement si R est atteignable avec l'un des 4 ensembles P_{n-1}. A chaque construction d'un nouvel ensemble P_{n-k}, on vérifie si R appartient à P_{n-k}. Si R n'appartient à aucun ensemble P_0 constructible à partir de P_n, alors il n'y a pas de solution.

Exemple simple avec 3 plaques :

Considérons le résultat $R = 120$ à atteindre à partir de l'ensemble $P_3 = \{2; 10; 100\}$.

A partir de cet ensemble P_3, il est possible de construire les couples suivants : $couples(P_3) = \{(2, 10); (2, 100); (10, 100)\}$.

Considérons le premier couple : $(2, 10)$.

En appliquant l'addition sur ce premier couple, on peut construire un premier ensemble P_2 égale à $\{12, 100\}$. Il faut donc tester si R peut être obtenu à partir de cet ensemble P_2. Un seul couple peut être obtenu à partir de P_2 : $(12, 100)$. En appliquant l'addition, on obtient un ensemble $P_3 = \{112\}$ dont le seul élément est différent de R. En appliquant la multiplication, on obtient un autre ensemble : $P_3 = \{1200\}$ dont le seul élément est aussi différent de R. En appliquant la soustraction $(120-12)$, on obtient un troisième ensemble $P_3 = \{88\}$ ne contenant pas R. La division n'est pas applicable puisque le reste de $100/12$ est non nul.

Cela signifie que le premier ensemble P_2 construit à partir de l'addition et du couple $(2, 10)$ ne permet pas d'obtenir R. Appliquons donc la multiplication pour construire un second ensemble $P_2 = \{20, 100\}$. Ici encore, un seul couple peut être obtenu à partir de P_2 : $(20, 100)$. En appliquant l'addition, on obtient un ensemble $P_3 = \{120\}$ contenant le résultat R recherché. L'algorithme s'arrête donc et renvoie la solution : $(2 \times 10) + 100$.

Sujet : Écrire un programme permettant de résoudre le jeu « Le compte est bon » à l'aide de l'algorithme précédemment décrit.

Les questions suivantes permettent de vous aider dans l'écriture de votre programme :

1. Écrire une fonction permettant de construire tous les couples (p_i, p_j) de P_n.
2. Écrire une fonction permettant de construire les quatre ensembles possibles P_{n-1} à partir de P_n.
3. Écrire la fonction d'exploration vérifiant si un résultat R est atteignable ou non à partir de P_n.
4. Modifier votre fonction de manière à retourner également la ligne de calcul permettant d'atteindre le résultat R.
5. Écrire une fonction permettant de tirer au hasard 6 plaques et une fonction renvoyant de manière aléatoire un résultat entre 100 et 999.

Analyse de l'algorithme : Nous vous proposons de structurer votre réflexion en fonction des réponses que vous allez fournir pour ces questions

1. Calculer le nombre d'ensembles examinés par l'algorithme dans le pire des cas en fonction de n. Combien y a-t-il d'ensembles lorsque $n = 6$?
2. Déterminer la complexité de l'algorithme d'exploration.
3. Déterminer le temps d'exécution moyen de l'algorithme utilisé pour résoudre le jeu "le compte est bon". (Faire une moyenne sur plusieurs mesures).

Question bonus : Ce problème est-il NP-complet ?

A.5.3 fiche de TP no 3 (optionnel)

L'objectif est d'implémenter les algorithmes de tri

— l'algorithme de tri par sélection

— l'algorithme de tri par insertion

— l'algorithme de tri à bulles

— l'algorithme de tri fusion

Évaluations temporelles

— Paramétrer votre fonction de manière à être capable de faire varier la taille du tableau. Nous pouvons admettre qu'une taille de 10^3 éléments est dite *petite*. La taille dite *moyenne* est définie telle que le tri par insertion soit 10 fois plus lent que pour une *petite* taille. La taille dite *grande* est quant à elle, définie telle que le tri par insertion soit 100 fois plus lent que pour une taille *moyenne*. Vous préciserez bien évidemment votre taille *moyenne* et votre taille *grande*. - Indiquer clairement le choix de ces tailles de tableau et fixer les pour chaque algorithme

Analyse des résultats

— Mesurez les temps d'exécution pour chaque algorithme précédemment introduits

— Tracez pour chaque algorithme les courbes des temps d'exécution.

— Commentez en quelques lignes vos résultats : comparez par exemple vos résultats avec les complexités théoriques vues en cours, indiquez s'il s'agit (i) du cas optimal, (ii) du pire des cas ou (iii) du cas moyen.