Complexité des Algorithmes et des Problèmes

P, NP et NP-Complétude

Moustapha Diaby
ESATIC

5 octobre 2026

Table des matières

1 Introduction **5**

2 Notion de problème et d'instance **6**
- 2.1 Notion de problème . . . 6
- 2.2 Instance d'un problème . . . 6
- 2.3 Taille d'une instance . . . 6
- 2.4 Problèmes de décision et lien avec les problèmes d'optimisation . . . 6
  - 2.4.1 Pourquoi les problèmes de décision ? . . . 7
  - 2.4.2 Lien avec les problèmes d'optimisation . . . 7
  - 2.4.3 Transformation optimisation → décision . . . 7
  - 2.4.4 Importance de cette transformation . . . 7
  - 2.4.5 Lien avec la classe NP . . . 7
  - 2.4.6 Conclusion . . . 8

3 Types de complexité et modes d'analyse **8**
- 3.1 Complexité temporelle . . . 8
- 3.2 Complexité spatiale . . . 8
- 3.3 Lien entre complexité temporelle et spatiale . . . 8
- 3.4 Modes d'analyse de la complexité . . . 8
  - 3.4.1 Meilleur cas . . . 9
  - 3.4.2 Pire cas . . . 9
  - 3.4.3 Cas moyen . . . 9
- 3.5 Choix du pire cas . . . 9
- 3.6 Positionnement de ce document . . . 9
- 3.7 Notations asymptotiques . . . 10
  - 3.7.1 Notation grand O . . . 10
  - 3.7.2 Notation grand Omega . . . 10
  - 3.7.3 Notation grand Theta . . . 10
  - 3.7.4 Comparaison des notations . . . 11
  - 3.7.5 Exemples classiques . . . 11
  - 3.7.6 Ordre de croissance des fonctions . . . 11
  - 3.7.7 Lien avec l'analyse des algorithmes . . . 11

4 Calcul effectif de la complexité des algorithmes **11**
- 4.1 Opérations élémentaires . . . 11

4.2 Instructions séquentielles . . . 12
4.3 Structures conditionnelles . . . 12
4.4 Boucles . . . 12
4.5 Boucles imbriquées . . . 13
4.6 Exemple : recherche séquentielle . . . 13
4.7 Exemple : tri à bulle . . . 13
4.8 Complexité des algorithmes récursifs . . . 13
4.8.1 Introduction . . . 13
4.8.2 Relation de récurrence . . . 14
4.8.3 Exemple simple : somme récursive . . . 14
4.8.4 Exemple : recherche binaire . . . 14
4.8.5 Méthodes de résolution des récurrences . . . 15
4.8.6 3. Méthode de l'arbre de récursion . . . 15
4.8.7 Théorème fondamental : Théorème maître . . . 15
4.8.8 Exemples d'application du théorème maître . . . 15
4.8.9 Exemple exponentiel : Fibonacci . . . 16
4.8.10 Analyse par arbre de récursion . . . 16
4.8.11 Comparaison des complexités récursives . . . 16
4.8.12 Remarques . . . 16
4.9 Classe de complexité . . . 16
4.10 Explosion combinatoire : exemple du voyageur de commerce . . . 17

5 Complexité des problèmes 18
5.1 Bornes supérieure et inférieure . . . 18
5.1.1 Borne supérieure . . . 18
5.1.2 Borne inférieure . . . 18
5.2 Méthodes pour établir des bornes . . . 18

6 Réduction polynomiale 18
6.1 Principe . . . 19
6.2 Définition formelle . . . 19
6.3 Interprétation . . . 19
6.4 Utilisation . . . 19

7 Méthode de l'oracle 19
7.1 Principe . . . 19
7.2 Utilisation . . . 19
7.3 Intérêt . . . 20

8 Arbres de décision 20
8.1 Principe . . . 20
8.2 Profondeur de l'arbre . . . 20
8.3 Exemple : tri par comparaison . . . 20

9 Complexité polynomiale 21
9.1 Exemples . . . 21

10 Machines de Turing 21

11 La classe P 21
11.1 Définition . . . 21
11.2 Interprétation . . . 21
11.3 Exemples de problèmes appartenant à P . . . 22

11.3.1 Tri . . . 22
11.3.2 Recherche dans un tableau . . . 22
11.3.3 Plus court chemin . . . 22
11.3.4 Arbre couvrant minimum . . . 22
11.3.5 Multiplication de matrices . . . 22
11.4 Caractéristiques des problèmes de P . . . 23
11.5 Problèmes de décision . . . 23
11.6 Stabilité de la classe P . . . 23

12 La classe NP **23**
12.1 Définition . . . 23
12.2 Notion de certificat . . . 24
12.3 Interprétation intuitive . . . 24
12.4 Machine de Turing non déterministe . . . 24
12.5 Relation entre P et NP . . . 24
12.6 Structure de la classe NP . . . 24
12.7 Exemples supplémentaires . . . 25

13 Le problème SAT **25**
13.1 Variables booléennes . . . 25
13.2 Littéraux . . . 25
13.3 Clause . . . 25
13.4 Formule booléenne . . . 26
13.5 Forme normale conjonctive . . . 26
13.6 Exemple de formule CNF . . . 26
13.7 Affectation des variables . . . 26
13.8 Exemple détaillé . . . 26
13.9 Définition du problème SAT . . . 27
13.10 Vérification en temps polynomial . . . 27
13.11 Problème 3-SAT . . . 27
13.12 Théorème de Cook-Levin . . . 27
13.13 Idée de la preuve . . . 27
13.14 Importance du problème SAT . . . 28
13.15 Applications concrètes de quelques problèmes de NP . . . 28
13.15.1 SAT . . . 28
13.15.2 3-SAT . . . 28
13.15.3 CLIQUE . . . 28
13.15.4 Vertex Cover . . . 29
13.15.5 Cycle Hamiltonien . . . 29

14 Les problèmes NP-complets **29**
14.1 Définition . . . 29
14.2 Problèmes NP-difficiles . . . 30
14.3 Relation entre les classes . . . 30
14.4 Importance des problèmes NP-complets . . . 30
14.5 Premier problème NP-complet . . . 30
14.6 Méthode pour démontrer la NP-complétude . . . 30
14.7 Exemples célèbres de problèmes NP-complets . . . 31
14.8 Chaîne de réductions . . . 31
14.9 Le problème du voyageur de commerce . . . 31
14.10 Conséquences de P = NP . . . 32
14.11 Approches pour traiter les problèmes NP-complets . . . 32

15 Réduction polynomiale : exemple détaillé **33**
15.1 Rappel du problème 3-SAT 33
15.2 Rappel du problème CLIQUE 33
15.3 Idée de la réduction 33
15.4 Construction du graphe 33
15.5 Exemple 34
15.6 Interprétation 34
15.7 Correction de la réduction 34
15.8 Complexité de la transformation 34
15.9 Conclusion 34

16 Conclusion **35**

17 Exercices corrigés **35**

18 Exercices corrigés **37**
18.1 Exercice 4 41

1 Introduction

La théorie de la complexité algorithmique constitue un domaine fondamental de l'informatique théorique. Elle vise à classifier les problèmes informatiques selon les ressources nécessaires pour les résoudre.

Lorsqu'un problème doit être résolu par un ordinateur, il est nécessaire de concevoir un algorithme capable de produire la solution à partir des données d'entrée. Toutefois, tous les algorithmes ne possèdent pas la même efficacité. Certains problèmes peuvent être résolus rapidement même pour de grandes tailles d'entrée, tandis que d'autres deviennent extrêmement difficiles dès que la taille du problème augmente.

La théorie de la complexité cherche donc à répondre à des questions fondamentales :

— combien de temps est nécessaire pour résoudre un problème ?
— quelle quantité de mémoire est requise ?
— existe-t-il un algorithme **efficace** pour ce problème ?
— certains problèmes sont-ils intrinsèquement difficiles ?

Pour répondre à ces questions, la théorie de la complexité analyse les ressources utilisées par les algorithmes.

Les deux ressources principales étudiées sont :

— le temps de calcul
— l'espace mémoire

Dans ce document, nous nous intéressons principalement à la **complexité temporelle**, c'est-à-dire au nombre d'opérations nécessaires pour résoudre un problème.

Une notion importante dans l'étude de la complexité est la taille de l'entrée du problème, généralement notée n. La complexité d'un algorithme est alors exprimée comme une fonction de n.

Afin d'obtenir des garanties solides sur les performances des algorithmes, l'analyse est généralement effectuée dans le **pire des cas**. Cette approche permet de déterminer une limite supérieure sur le temps d'exécution et d'assurer que l'algorithme reste efficace quelles que soient les données d'entrée.

La théorie de la complexité ne s'intéresse pas uniquement à l'étude des algorithmes, mais également à la difficulté intrinsèque des problèmes eux-mêmes. Deux problèmes différents peuvent nécessiter des quantités très différentes de ressources pour être résolus.

Afin de mieux comprendre cette difficulté, les problèmes sont regroupés en différentes classes de complexité. Parmi les plus importantes figurent :

— la classe P, qui contient les problèmes pouvant être résolus en temps polynomial
— la classe NP, qui contient les problèmes dont une solution peut être vérifiée en temps polynomial
— les problèmes NP-complets, qui sont les problèmes les plus difficiles de la classe NP

L'étude de ces classes permet de mieux comprendre les limites des algorithmes et les relations entre les différents problèmes informatiques.

L'une des questions les plus célèbres de l'informatique théorique est le problème :

$$P = NP?$$

Cette question demande si tous les problèmes dont la solution peut être vérifiée rapidement peuvent également être résolus rapidement.

Malgré plusieurs décennies de recherche, cette question reste ouverte et constitue l'un des plus grands défis de la science informatique.

2 Notion de problème et d'instance

Avant d'étudier la complexité des algorithmes, il est nécessaire de distinguer clairement les notions de **problème** et d'**instance** d'un problème.

2.1 Notion de problème

En informatique, un problème correspond à une question générale pour laquelle on cherche une méthode systématique permettant de produire une solution.

Un problème décrit généralement :

— les données d'entrée possibles

— les propriétés que doit satisfaire la solution

Un algorithme est alors une procédure permettant de résoudre ce problème pour toutes les entrées possibles.

Exemple

Considérons le problème suivant :

Déterminer le plus grand élément d'un tableau d'entiers.

Ce problème est une description générale et ne correspond pas à une entrée particulière.

2.2 Instance d'un problème

Une **instance** d'un problème correspond à une configuration particulière des données d'entrée. Autrement dit, une instance est un cas concret du problème.

Exemple

Pour le problème précédent, les tableaux suivants sont des instances possibles :

$$[3, 7, 2, 9, 5]$$

$$[12, 4, 18, 6]$$

Chaque tableau représente une instance différente du même problème.

2.3 Taille d'une instance

La taille d'une instance correspond à une mesure de la quantité de données d'entrée.

Elle est généralement notée n.

Selon le type de problème, la taille peut correspondre à :

— le nombre d'éléments d'un tableau

— le nombre de sommets d'un graphe

— la longueur d'une chaîne de caractères

L'analyse de complexité consiste à étudier comment le temps d'exécution d'un algorithme évolue en fonction de cette taille n.

2.4 Problèmes de décision et lien avec les problèmes d'optimisation

Dans la théorie de la complexité, on étudie principalement les **problèmes de décision**.

Un **problème de décision** est un problème dont la réponse est uniquement :

oui ou non

Exemples :

— Un graphe est-il connexe ?

— Existe-t-il un chemin entre deux sommets ?

— Une formule booléenne est-elle satisfiable ?
— Existe-t-il une clique de taille k dans un graphe ?

2.4.1 Pourquoi les problèmes de décision ?

Les problèmes de décision sont privilégiés en théorie de la complexité car ils permettent une formalisation mathématique rigoureuse des classes de complexité telles que P, NP et les problèmes NP-complets.

2.4.2 Lien avec les problèmes d'optimisation

De nombreux problèmes étudiés en pratique sont des **problèmes d'optimisation**, c'est-à-dire des problèmes où l'on cherche la meilleure solution selon un critère donné (minimisation ou maximisation).

Exemples de problèmes d'optimisation :
— Trouver le plus court chemin dans un graphe
— Minimiser le coût d'un circuit (TSP)
— Maximiser la valeur dans le problème du sac à dos
Cependant, tout problème d'optimisation peut être transformé en problème de décision.

2.4.3 Transformation optimisation → décision

Le principe consiste à poser une question du type :

$$\text{Existe-t-il une solution de coût } \leq k \ ?$$

ou

$$\text{Existe-t-il une solution de valeur } \geq k \ ?$$

Exemple : plus court chemin
— *Problème d'optimisation :* trouver le plus court chemin entre deux sommets
— *Version décision :* existe-t-il un chemin de longueur $\leq k$?
Exemple : problème du sac à dos
— *Optimisation :* maximiser la valeur totale
— *Décision :* existe-t-il une solution de valeur $\geq V$?

2.4.4 Importance de cette transformation

Cette transformation est fondamentale car :
— les classes de complexité (P, NP, NP-complet) sont définies sur les problèmes de décision
— elle permet d'analyser la difficulté des problèmes d'optimisation
— elle est essentielle dans les preuves de NP-complétude

2.4.5 Lien avec la classe NP

Un problème appartient à la classe NP si une solution candidate peut être vérifiée en temps polynomial.

Dans les problèmes d'optimisation :
— il est souvent difficile de trouver la solution optimale
— mais il est facile de vérifier une solution donnée
Exemple :
Dans le problème du voyageur de commerce :
— trouver le meilleur circuit est difficile
— vérifier qu'un circuit donné a une longueur $\leq k$ est facile

2.4.6 Conclusion

Ainsi, bien que les problèmes d'optimisation soient omniprésents en pratique, la théorie de la complexité repose essentiellement sur les problèmes de décision. La transformation d'un problème d'optimisation en problème de décision constitue donc un outil central pour analyser la difficulté des problèmes et établir des résultats fondamentaux en complexité algorithmique.

3 Types de complexité et modes d'analyse

L'étude de la complexité d'un algorithme consiste à mesurer les ressources nécessaires pour résoudre un problème en fonction de la taille de l'entrée.

Deux ressources sont généralement considérées :

— le temps de calcul

— l'espace mémoire

3.1 Complexité temporelle

La complexité temporelle mesure le nombre d'opérations élémentaires effectuées par un algorithme en fonction de la taille de l'entrée n. Elle permet d'évaluer la vitesse de croissance du temps d'exécution.

Exemple :

$$T(n) = 2n^2 + 3n + 5$$

Dans l'analyse asymptotique, les constantes sont ignorées et l'on retient le terme dominant :

$$T(n) = O(n^2)$$

3.2 Complexité spatiale

La complexité spatiale mesure la quantité de mémoire utilisée par l'algorithme pendant son exécution.

Elle inclut :

— la mémoire pour les variables

— les structures de données

— la mémoire auxiliaire utilisée par l'algorithme

Par exemple, un algorithme de tri peut nécessiter :

— $O(1)$ mémoire supplémentaire (tri en place)

— $O(n)$ mémoire supplémentaire (tri fusion)

3.3 Lien entre complexité temporelle et spatiale

Dans de nombreux algorithmes, il existe un compromis entre le temps de calcul et l'espace mémoire. On parle de **compromis temps-mémoire**.

Par exemple :

— utiliser plus de mémoire peut permettre d'accélérer les calculs

— réduire l'utilisation de mémoire peut augmenter le temps d'exécution

Un exemple classique est l'utilisation de tables de hachage ou de mémorisation (memoization) en programmation dynamique.

3.4 Modes d'analyse de la complexité

La complexité d'un algorithme peut être étudiée selon trois situations différentes.

3.4.1 Meilleur cas

Le meilleur cas correspond à la situation la plus favorable. L'algorithme effectue le nombre minimal d'opérations.

Exemple : recherche d'un élément dans un tableau.
Si l'élément recherché est au début du tableau :

$$T(n) = O(1)$$

3.4.2 Pire cas

Le pire cas correspond à la situation la plus défavorable. L'algorithme effectue le nombre maximal d'opérations.

Exemple : recherche dans un tableau non trié.
Si l'élément est à la fin ou absent :

$$T(n) = O(n)$$

3.4.3 Cas moyen

Le cas moyen correspond à la complexité moyenne sur l'ensemble des entrées possibles. Il nécessite généralement une analyse probabiliste.
Par exemple, pour la recherche dans un tableau :

$$T(n) = O(n)$$

en moyenne.

3.5 Choix du pire cas

Dans la théorie de la complexité, l'analyse est généralement effectuée dans le **pire des cas**.
Cela permet :
— d'obtenir une borne garantie sur le temps d'exécution
— de comparer les algorithmes de manière rigoureuse
— d'éviter les hypothèses probabilistes sur les données

3.6 Positionnement de ce document

Dans ce document, comme annoncé dans l'introduction nous nous intéressons principalement à :
— la complexité temporelle
— dans le pire des cas
Cette approche est celle utilisée dans la théorie de la complexité et dans l'étude des classes de problèmes telles que :
— la classe P
— la classe NP
— les problèmes NP-complets

3.7 Notations asymptotiques

Pour analyser la complexité des algorithmes, on utilise des notations asymptotiques permettant d'étudier la croissance du temps d'exécution lorsque la taille de l'entrée n devient très grande.

Ces notations permettent d'ignorer :

— les constantes multiplicatives

— les termes de faible ordre

afin de se concentrer uniquement sur le comportement asymptotique dominant.

3.7.1 Notation grand O

La notation O donne une **borne supérieure** sur la complexité.

Définition

Une fonction $f(n)$ est $O(g(n))$ s'il existe deux constantes $c > 0$ et n_0 telles que :

$$f(n) \leq c\, g(n) \quad \text{pour tout } n \geq n_0$$

Interprétation

La fonction $f(n)$ ne croît pas plus vite que $g(n)$ à un facteur constant près.

Exemple

$$3n^2 + 2n + 1 = O(n^2)$$

car le terme dominant est n^2.

3.7.2 Notation grand Omega

La notation Ω donne une **borne inférieure**.

Définition

Une fonction $f(n)$ est $\Omega(g(n))$ s'il existe deux constantes $c > 0$ et n_0 telles que :

$$f(n) \geq c\, g(n) \quad \text{pour tout } n \geq n_0$$

Interprétation

La fonction croît au moins aussi vite que $g(n)$.

Exemple

$$3n^2 + 2n + 1 = \Omega(n^2)$$

3.7.3 Notation grand Theta

La notation Θ donne une **borne asymptotique exacte**.

Définition

Une fonction $f(n)$ est $\Theta(g(n))$ si :

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

Autrement dit :

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

pour certaines constantes positives c_1, c_2 et n_0.

Exemple

$$3n^2 + 2n + 1 = \Theta(n^2)$$

3.7.4 Comparaison des notations

Notation	Signification
$O(g(n))$	borne supérieure
$\Omega(g(n))$	borne inférieure
$\Theta(g(n))$	borne asymptotique exacte

3.7.5 Exemples classiques

Fonction	Complexité
$n + 10$	$\Theta(n)$
$n^2 + 5n$	$\Theta(n^2)$
$n \log n + n$	$\Theta(n \log n)$
$2^n + n^3$	$\Theta(2^n)$

3.7.6 Ordre de croissance des fonctions

Les fonctions suivantes sont classées selon leur croissance :

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

Les algorithmes exponentiels deviennent rapidement impraticables lorsque n augmente.

3.7.7 Lien avec l'analyse des algorithmes

Dans ce cours, la notation O est principalement utilisée pour exprimer la complexité temporelle dans le pire des cas.

Cependant, les notations Ω et Θ permettent d'obtenir une analyse plus précise du comportement des algorithmes.

4 Calcul effectif de la complexité des algorithmes

Pour analyser la complexité d'un algorithme, on cherche à estimer le nombre d'opérations élémentaires exécutées en fonction de la taille des données d'entrée.

On note généralement :

$$n : \text{taille des données}$$

$$T(n) : \text{nombre d'opérations élémentaires}$$

L'objectif est donc de déterminer une fonction $T(n)$ qui mesure le coût de l'algorithme.

4.1 Opérations élémentaires

Une opération élémentaire est une instruction simple dont le temps d'exécution est considéré comme constant. On considère généralement comme opérations élémentaires :

— les opérations arithmétiques
— les comparaisons
— les affectations
— l'accès à un élément d'un tableau

Ces opérations ont un coût constant :

$$O(1)$$

4.2 Instructions séquentielles

Lorsque plusieurs instructions sont exécutées successivement, la complexité totale est la somme des coûts de chaque instruction.

$$T(n) = T_1(n) + T_2(n)$$

Exemple

```
a = b + c
d = a * 2
```

Chaque instruction ayant un coût constant, la complexité totale reste :

$$O(1)$$

4.3 Structures conditionnelles

Dans une structure conditionnelle, la complexité correspond au coût du test (négligeable très souvent) plus la complexité maximale des blocs exécutés.

$$T(n) = \max(T_1(n), T_2(n))$$

Exemple

```
if x > y:
   a = x
else
   a = y
```

La complexité reste constante :

$$O(1)$$

4.4 Boucles

Pour une boucle, la complexité correspond à la somme des coûts de chaque itération.
Exemple

```
for i = 1 to n
   somme = somme + A[i]
```

La boucle est exécutée n fois.

$$T(n) = n$$

La complexité est donc :

$$O(n)$$

4.5 Boucles imbriquées

Lorsque plusieurs boucles sont imbriquées, les nombres d'itérations se multiplient.
Exemple

```
for i = 1 to n
   for j = 1 to n
       print(i,j)
```

Le nombre total d'itérations est :

$$n \times n = n^2$$

La complexité est :

$$O(n^2)$$

4.6 Exemple : recherche séquentielle

Considérons l'algorithme suivant :

```
i = 1
while i  n and A[i]  x
   i = i + 1
```

Dans le pire des cas, la boucle est exécutée n fois.

$$T(n) = O(n)$$

4.7 Exemple : tri à bulle

Le tri à bulle utilise deux boucles imbriquées.

$$T(n) = \sum_{i=1}^{n-1} i$$

ce qui donne :

$$T(n) = \frac{n(n-1)}{2}$$

La complexité est donc :

$$O(n^2)$$

4.8 Complexité des algorithmes récursifs

4.8.1 Introduction

Les algorithmes récursifs sont définis en termes d'eux-mêmes : un problème de taille n est résolu en le décomposant en sous-problèmes de taille plus petite.

L'analyse de leur complexité repose généralement sur l'établissement d'une **relation de récurrence** qui exprime le temps d'exécution en fonction de la taille de l'entrée.

4.8.2 Relation de récurrence

Une relation de récurrence est une équation qui définit le temps d'exécution $T(n)$ d'un algorithme récursif en fonction de T appliqué à des tailles plus petites.

Exemple :

$$T(n) = T(n-1) + c$$

ou

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

4.8.3 Exemple simple : somme récursive

```
def somme(n):
    if n == 0:
        return 0
    return n + somme(n-1)
```

Relation de récurrence :

$$T(n) = T(n-1) + c$$

Développement :

$$T(n) = T(n-1) + c = T(n-2) + 2c = \cdots = T(0) + nc$$

Donc :

$$T(n) = O(n)$$

4.8.4 Exemple : recherche binaire

```
def recherche_binaire(tab, x):
    if len(tab) == 0:
        return False
    milieu = len(tab)//2
    if tab[milieu] == x:
        return True
    elif x < tab[milieu]:
        return recherche_binaire(tab[:milieu], x)
    else:
        return recherche_binaire(tab[milieu+1:], x)
```

Relation :

$$T(n) = T(n/2) + c$$

Développement :

$$T(n) = T(n/2) + c = T(n/4) + 2c = \cdots = T(1) + c\log n$$

Donc :

$$T(n) = O(\log n)$$

4.8.5 Méthodes de résolution des récurrences

1. Méthode de substitution
On suppose une forme pour $T(n)$ puis on la vérifie par récurrence.
2. Méthode d'expansion (itération)
On développe la récurrence jusqu'au cas de base.

4.8.6 3. Méthode de l'arbre de récursion

On représente les appels récursifs sous forme d'arbre et on somme les coûts.

4.8.7 Théorème fondamental : Théorème maître

[Théorème maître] Considérons une relation de la forme :

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

où :
— $a \geq 1$: nombre de sous-problèmes
— $b > 1$: facteur de réduction
— $f(n)$: coût du travail hors récursion
Alors :

1. Si $f(n) = O(n^{\log_b a - \epsilon})$ pour un $\epsilon > 0$:

$$T(n) = \Theta(n^{\log_b a})$$

2. Si $f(n) = \Theta(n^{\log_b a})$:

$$T(n) = \Theta(n^{\log_b a} \log n)$$

3. Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ et condition de régularité :

$$T(n) = \Theta(f(n))$$

4.8.8 Exemples d'application du théorème maître

Exemple 1 : tri fusion

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

Ici :
— $a = 2$
— $b = 2$
— $f(n) = n$
— $n^{\log_2 2} = n$
Donc cas 2 :

$$T(n) = \Theta(n \log n)$$

Exemple 2 : recherche binaire

$$T(n) = T(n/2) + c$$

$$T(n) = O(\log n)$$

4.8.9 Exemple exponentiel : Fibonacci

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

Relation :

$$T(n) = T(n-1) + T(n-2) + c$$

Croissance :

$$T(n) = O(\varphi^n)$$

avec $\varphi \approx 1.618$.
Conclusion : complexité exponentielle.

4.8.10 Analyse par arbre de récursion

Pour Fibonacci :
— Nombre de nœuds $\approx 2^n$
— Chaque appel fait un travail constant
Donc :

$$T(n) = O(2^n)$$

4.8.11 Comparaison des complexités récursives

Type de récurrence	Complexité
$T(n) = T(n-1) + c$	$O(n)$
$T(n) = T(n/2) + c$	$O(\log n)$
$T(n) = 2T(n/2) + n$	$O(n \log n)$
$T(n) = 2T(n/2) + c$	$O(n)$
$T(n) = T(n-1) + T(n-2)$	$O(2^n)$

4.8.12 Remarques

— La récursivité peut entraîner une explosion combinatoire.
— Les algorithmes récursifs sont souvent élégants mais parfois inefficaces.
— L'optimisation passe par :
  — la mémoïsation
  — la programmation dynamique

4.9 Classe de complexité

Classe de complexité	Description	Exemples d'algorithmes
$O(1)$	Temps constant	Accès à un élément d'un tableau
$O(\log n)$	Temps logarithmique	Recherche dichotomique
$O(n)$	Temps linéaire	Recherche séquentielle dans un tableau
$O(n \log n)$	Temps quasi-linéaire	Tri fusion, tri rapide (cas moyen)
$O(n^2)$	Temps quadratique	Tri à bulle, tri par insertion
$O(n^3)$	Temps cubique	Multiplication naïve de matrices
$O(2^n)$	Temps exponentiel	Recherche exhaustive, problèmes combinatoires
$O(n!)$	Temps factoriel	Génération de toutes les permutations

4.10 Explosion combinatoire : exemple du voyageur de commerce

Le problème du voyageur de commerce (TSP) illustre parfaitement les limites des méthodes de recherche exhaustive. La méthode la plus simple consiste à tester toutes les permutations possibles des villes. Pour n villes, le nombre de circuits possibles est :

$$(n-1)!$$

La complexité de l'algorithme naïf (Nous verrons le calcul plus loin !) est donc :

$$O(n!)$$

Hypothèse de calcul Supposons un ordinateur capable d'exécuter **trois instructions par nanoseconde**.

Nous considérons ensuite une machine théorique encore plus rapide, capable d'exécuter **une instruction par temps de Planck** (5.39×10^{-44} secondes), ce qui représente une limite physique extrêmement optimiste.

Temps de calcul théorique Le tableau suivant montre le temps nécessaire pour tester toutes les solutions possibles.

Nombre de villes	Ordinateur classique	Ordinateur au temps de Planck
10	≈ 0.01 s	instantané
15	≈ 1 heure	≈ 5 ms
19	≈ 1 an	≈ 1 s
27	$8\times$ âge de l'univers	≈ 12 h
40	4×10^{31} fois l'âge de l'univers	≈ 12 h
50	1.5×10^{48} fois l'âge de l'univers	$4000\times$ âge de l'univers

Interprétation Ces résultats illustrent un phénomène fondamental appelé **explosion combinatoire**.

Lorsque la taille du problème augmente légèrement, le nombre de solutions possibles croît de manière **factorielle**. Même avec une machine théorique extrêmement rapide, le temps de calcul devient rapidement inabordable.

Conclusion Ce type d'analyse montre que l'amélioration du matériel informatique ne suffit pas à résoudre certains problèmes. La conception d'**algorithmes plus efficaces** ou l'utilisation de **méthodes heuristiques** devient alors indispensable.

5 Complexité des problèmes

La complexité d'un problème correspond à la quantité minimale de ressources nécessaires pour le résoudre.

Dans la théorie de la complexité, on mesure généralement la complexité d'un problème à partir de la complexité des algorithmes qui le résolvent. Plus précisément, la complexité d'un problème est définie comme la complexité dans le **pire des cas** du meilleur algorithme capable de le résoudre.

Ainsi, si un algorithme résout un problème avec une complexité temporelle :

$$T(n)$$

alors cette fonction fournit une **borne supérieure** sur la complexité du problème.

5.1 Bornes supérieure et inférieure

Pour analyser la complexité d'un problème, on cherche généralement à déterminer deux bornes :

— une **borne supérieure**
— une **borne inférieure**

5.1.1 Borne supérieure

Une borne supérieure correspond à la complexité d'un algorithme qui résout le problème.

Si un algorithme résout un problème en temps :

$$T(n) = O(f(n))$$

alors la complexité du problème est au plus $O(f(n))$.

5.1.2 Borne inférieure

Une borne inférieure indique que tout algorithme résolvant le problème nécessite au moins un certain nombre d'opérations.

Elle s'exprime sous la forme :

$$T(n) = \Omega(g(n))$$

Déterminer une borne inférieure est généralement plus difficile car il faut prouver qu'aucun algorithme ne peut être plus rapide.

5.2 Méthodes pour établir des bornes

Plusieurs techniques sont utilisées pour établir les bornes de complexité des problèmes. Parmi les méthodes les plus importantes :

— la réduction polynomiale
— la méthode de l'oracle
— les arbres de décision

Ces méthodes permettent de comparer les problèmes et de démontrer des résultats de complexité. Dans les sections suivantes, nous présentons ces trois techniques.

6 Réduction polynomiale

La réduction polynomiale est l'un des outils les plus puissants de la théorie de la complexité.

Elle permet de comparer la difficulté de deux problèmes.

6.1 Principe

Supposons que nous connaissions un problème A dont la complexité est bien comprise. Si nous pouvons transformer toute instance de A en une instance d'un autre problème B en temps polynomial, alors résoudre B permet également de résoudre A.

On note cette relation :

$$A \leq_p B$$

ce qui signifie que A se réduit polynomialement à B.

6.2 Définition formelle

On dit que A se réduit polynomialement à B s'il existe une fonction de transformation f telle que :

— f est calculable en temps polynomial
— pour toute instance x de A, $f(x)$ est une instance de B
— x est solution de A si et seulement si $f(x)$ est solution de B

6.3 Interprétation

Si

$$A \leq_p B$$

alors B est au moins aussi difficile que A.

Par conséquent :

si B possède un algorithme polynomial, alors A possède également un algorithme polynomial.

6.4 Utilisation

La réduction polynomiale est principalement utilisée pour :
— démontrer qu'un problème est NP-complet
— comparer la difficulté de deux problèmes
— transférer des résultats de complexité

7 Méthode de l'oracle

La méthode de l'oracle est une technique conceptuelle utilisée en théorie de la complexité pour étudier les relations entre problèmes.

7.1 Principe

Un **oracle** est une boîte noire capable de résoudre instantanément un problème donné. On suppose que cet oracle peut répondre correctement à toute requête concernant ce problème.

7.2 Utilisation

Pour étudier la complexité d'un problème A, on suppose l'existence d'un oracle capable de résoudre un problème B. On analyse ensuite si cet oracle permet de résoudre A efficacement.

Si c'est le cas, on peut conclure que :

$$A \leq_p B$$

7.3 Intérêt

La méthode de l'oracle permet :
— d'étudier les relations entre classes de complexité
— d'analyser la puissance relative des problèmes
— de construire des réductions conceptuelles
Cette méthode est particulièrement utilisée dans l'étude des classes de complexité avancées.

8 Arbres de décision

Les arbres de décision constituent une technique importante pour établir des bornes inférieures sur la complexité des algorithmes.

8.1 Principe

Un arbre de décision représente les différentes décisions prises par un algorithme au cours de son exécution. Chaque noeud interne correspond à une opération de comparaison ou de test.

Chaque branche correspond à un résultat possible de cette opération. Les feuilles de l'arbre représentent les solutions finales.

8.2 Profondeur de l'arbre

La profondeur maximale de l'arbre correspond au nombre maximal d'opérations nécessaires pour résoudre le problème.

Cette profondeur fournit une borne inférieure sur la complexité de tout algorithme utilisant ce modèle.

8.3 Exemple : tri par comparaison

Dans le cas du tri par comparaison, chaque noeud de l'arbre correspond à une comparaison entre deux éléments.

Le nombre de permutations possibles de n éléments est :

$$n!$$

Un arbre de décision capable de distinguer toutes les permutations doit donc posséder au moins $n!$ feuilles.

On peut montrer que :

$$\log(n!) = \Theta(n \log n)$$

Ainsi, tout algorithme de tri basé uniquement sur des comparaisons nécessite au moins :

$$\Omega(n \log n)$$

comparaisons dans le pire des cas.

Cette technique permet donc d'établir une borne inférieure fondamentale pour les algorithmes de tri par comparaison.

9 Complexité polynomiale

Un algorithme est polynomial si :

$$T(n) = O(n^k)$$

où k est une constante.

9.1 Exemples

Algorithme	Complexité
Recherche linéaire	$O(n)$
Tri fusion	$O(n \log n)$
Dijkstra	$O(n^2)$
Floyd-Warshall	$O(n^3)$

10 Machines de Turing

La machine de Turing est le modèle théorique de calcul utilisé pour définir les classes de complexité.

Elle est composée de :

— une bande infinie
— une tête de lecture
— un ensemble d'états
— une fonction de transition

11 La classe P

La classe P occupe une place centrale dans la théorie de la complexité algorithmique. Elle correspond aux problèmes considérés comme **efficacement résolubles**.

11.1 Définition

La classe P est l'ensemble des problèmes de décision qui peuvent être résolus en temps polynomial par une machine de Turing déterministe.

Formellement :

$$P = \{L \mid L \text{ est décidable en temps polynomial}\}$$

Autrement dit, un problème appartient à P s'il existe un algorithme qui le résout avec une complexité :

$$T(n) = O(n^k)$$

où k est une constante et n la taille de l'entrée.

11.2 Interprétation

Les problèmes de la classe P sont généralement considérés comme **faciles** ou **tractables**.

Cela signifie que le temps de calcul croît de manière raisonnable lorsque la taille du problème augmente.

Par exemple :

$$n^2 \quad ou \quad n^3$$

restent gérables pour de grandes tailles d'entrée. En revanche, les complexités exponentielles comme :

$$2^n$$

deviennent rapidement impraticables.

11.3 Exemples de problèmes appartenant à P

De nombreux problèmes fondamentaux de l'informatique appartiennent à la classe P.

11.3.1 Tri

Le problème du tri consiste à ordonner un ensemble d'éléments.
Algorithmes connus :
— Merge Sort : $O(n \log n)$
— Heap Sort : $O(n \log n)$
— Quick Sort (cas moyen) : $O(n \log n)$

11.3.2 Recherche dans un tableau

— recherche linéaire : $O(n)$
— recherche dichotomique : $O(\log n)$

11.3.3 Plus court chemin

Dans un graphe pondéré, il s'agit de trouver le chemin de coût minimal entre deux sommets.
Algorithmes classiques :
— Dijkstra : $O(n^2)$
— Bellman-Ford : $O(nm)$

11.3.4 Arbre couvrant minimum

Dans un graphe pondéré, on cherche un arbre reliant tous les sommets avec un coût minimal.
Algorithmes :
— Kruskal : $O(m \log n)$
— Prim : $O(n^2)$

11.3.5 Multiplication de matrices

La multiplication de deux matrices peut être effectuée en temps polynomial.
Algorithme classique :

$$O(n^3)$$

Algorithme de Strassen :

$$O(n^{2.81})$$

11.4 Caractéristiques des problèmes de P

Les problèmes appartenant à la classe P présentent généralement les caractéristiques suivantes :
— existence d'un algorithme déterministe efficace
— complexité polynomiale
— croissance raisonnable du temps d'exécution

Ces problèmes peuvent être résolus de manière fiable même pour des instances relativement grandes.

11.5 Problèmes de décision

La théorie de la complexité se concentre principalement sur les **problèmes de décision**.
Un problème de décision est un problème dont la réponse est :

$$oui \quad ou \quad non$$

Exemple :
— existe-t-il un chemin entre deux sommets ?
— le graphe est-il connexe ?
— un nombre est-il premier ?

Même les problèmes d'optimisation peuvent être reformulés comme des problèmes de décision.
Exemple :
Problème d'optimisation
Trouver le plus court chemin.
Version décisionnelle
Existe-t-il un chemin de longueur $\leq k$?

11.6 Stabilité de la classe P

La classe P est stable par plusieurs opérations importantes.
Par exemple :
— composition d'algorithmes
— boucles polynomiales
— appels à des sous-algorithmes polynomiaux

Cela signifie que combiner plusieurs algorithmes polynomiaux produit encore un algorithme polynomial.

12 La classe NP

La classe NP est l'une des classes fondamentales de la théorie de la complexité algorithmique. Elle regroupe les problèmes pour lesquels une solution peut être vérifiée efficacement.

12.1 Définition

La classe NP (Non Deterministic Polynomial time) est l'ensemble des problèmes de décision pour lesquels une solution candidate peut être vérifiée en temps polynomial.
Formellement :

$$NP = \{L \mid \text{il existe un algorithme de vérification polynomial}\}$$

Autrement dit, si une solution est fournie, on peut vérifier sa validité en temps polynomial.

12.2 Notion de certificat

Pour les problèmes de la classe NP, on introduit la notion de **certificat**.

Un certificat est une information supplémentaire permettant de vérifier rapidement qu'une solution est correcte.

Formellement :

Un problème appartient à NP s'il existe :

— un certificat y

— un algorithme de vérification polynomial V

tel que :

$$x \in L \iff \exists y : V(x, y) = 1$$

où x représente l'instance du problème.

12.3 Interprétation intuitive

Les problèmes de NP sont des problèmes pour lesquels :

— il est difficile de trouver une solution

— mais facile de vérifier une solution

Cette distinction est fondamentale en informatique.

12.4 Machine de Turing non déterministe

Une autre manière de définir la classe NP est à l'aide des machines de Turing non déterministes. Une machine non déterministe peut effectuer plusieurs choix possibles à chaque étape de calcul. On peut voir ce modèle comme une machine capable d'explorer simultanément toutes les possibilités.

Un problème appartient à NP s'il peut être résolu en temps polynomial par une machine de Turing non déterministe.

12.5 Relation entre P et NP

La relation fondamentale est :

$$P \subseteq NP$$

En effet, si un problème peut être résolu en temps polynomial, alors sa solution peut également être vérifiée en temps polynomial.

La grande question ouverte est :

$$P = NP?$$

Si cette égalité est vraie, alors tous les problèmes de NP pourraient être résolus efficacement.

12.6 Structure de la classe NP

La classe NP contient plusieurs sous-ensembles importants :

— les problèmes de la classe P

— les problèmes NP-complets

— les problèmes NP-intermédiaires

Les problèmes NP-complets sont les plus difficiles de la classe NP.

Si l'un d'entre eux possède un algorithme polynomial, alors :

$$P = NP$$

12.7 Exemples supplémentaires

Quelques exemples célèbres de problèmes appartenant à NP :
— SAT
— 3-SAT
— CLIQUE
— Vertex Cover
— Hamiltonian Cycle
— Traveling Salesman Problem (version décision)
— Subset Sum

13 Le problème SAT

Le problème SAT (Satisfiability Problem) est l'un des problèmes fondamentaux de l'informatique théorique.

Il consiste à déterminer si une formule logique booléenne peut être rendue vraie en attribuant des valeurs aux variables.

SAT joue un rôle central dans la théorie de la complexité car il est le premier problème démontré **NP-complet** par le théorème de Cook-Levin.

13.1 Variables booléennes

Une variable booléenne est une variable qui peut prendre deux valeurs :

$$vrai \quad ou \quad faux$$

On note généralement les variables :

$$x_1, x_2, x_3, \ldots, x_n$$

13.2 Littéraux

Un **littéral** est soit :
— une variable booléenne
— la négation d'une variable
Exemples de littéraux :

$$x_1, \quad x_2, \quad \neg x_3$$

Ainsi :
— x_1 est un littéral positif
— $\neg x_3$ est un littéral négatif

13.3 Clause

Une **clause** est une disjonction (OU logique) de littéraux.
Exemple :

$$(x_1 \vee \neg x_2 \vee x_3)$$

Cette clause est satisfaite si au moins un des littéraux est vrai.

13.4 Formule booléenne

Une formule booléenne est construite à partir de variables, de connecteurs logiques et de parenthèses.

Les connecteurs principaux sont :
— $\vee$: OU logique
— $\wedge$: ET logique
— $\neg$: négation

13.5 Forme normale conjonctive

Dans l'étude du problème SAT, les formules sont souvent exprimées en **forme normale conjonctive (CNF)**.

Une formule est en CNF si elle est une conjonction de clauses.

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

où chaque C_i est une clause.

13.6 Exemple de formule CNF

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$$

Dans cet exemple :
— il y a 3 clauses
— les variables sont x_1, x_2, x_3

13.7 Affectation des variables

Une **affectation** consiste à attribuer une valeur à chaque variable.

Exemple :

$$x_1 = vrai, \quad x_2 = faux, \quad x_3 = vrai$$

On remplace ensuite les variables dans la formule pour vérifier si elle devient vraie.

13.8 Exemple détaillé

Considérons la formule :

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$$

Prenons l'affectation suivante :

$$x_1 = vrai, \quad x_2 = faux, \quad x_3 = faux$$

Première clause :

$$(vrai \vee vrai) = vrai$$

Deuxième clause :

$$(faux \vee faux) = faux$$

La formule devient :

$$vrai \wedge faux = faux$$

Cette affectation ne satisfait pas la formule.

13.9 Définition du problème SAT

Entrée
Une formule booléenne F.
Question
Existe-t-il une affectation des variables telle que :

$$F = vrai$$

Si oui, la formule est dite **satisfiable**.
Sinon elle est **insatisfiable**.

13.10 Vérification en temps polynomial

Si une affectation est fournie, il est facile de vérifier si elle satisfait la formule.
Il suffit de :
— évaluer chaque clause
— vérifier que toutes les clauses sont vraies
Cette vérification peut être effectuée en temps polynomial.
C'est pourquoi SAT appartient à la classe NP.

13.11 Problème 3-SAT

Une variante importante est le problème **3-SAT**.
Dans ce problème, chaque clause contient exactement trois littéraux.
Exemple :

$$(x_1 \vee x_2 \vee \neg x_3)$$

Une formule 3-SAT est donc de la forme :

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_4 \vee x_5) \wedge (x_2 \vee \neg x_3 \vee x_6)$$

Le problème 3-SAT est également **NP-complet**.

13.12 Théorème de Cook-Levin

Théorème
SAT est NP-complet.

13.13 Idée de la preuve

On montre que toute computation d'une machine de Turing peut être encodée sous forme de formule booléenne. Ainsi, toute computation en temps polynomial peut être transformée en instance de SAT.

13.14 Importance du problème SAT

Le problème SAT est fondamental car :
— il est le premier problème NP-complet
— de nombreux problèmes peuvent être réduits à SAT
— il sert de point de départ pour prouver la NP-complétude
Beaucoup de réductions classiques utilisent la chaîne suivante :

$$SAT \rightarrow 3SAT \rightarrow CLIQUE$$

13.15 Applications concrètes de quelques problèmes de NP

La classe NP est importante car elle contient de nombreux problèmes difficiles apparaissant dans :
— l'optimisation combinatoire
— la recherche opérationnelle
— l'intelligence artificielle
— la cryptographie
— la planification
Plusieurs problèmes appartenant à la classe NP apparaissent dans des applications réelles. Nous présentons ci-dessous quelques exemples célèbres.

13.15.1 SAT

Le problème SAT consiste à déterminer si une formule booléenne peut être rendue vraie.
Applications :
— **Vérification de circuits électroniques**
Dans la conception de microprocesseurs, on modélise le circuit par une formule logique. Le problème SAT permet de vérifier si certaines configurations peuvent produire une erreur.
— **Vérification de logiciels**
Les outils de model checking utilisent SAT pour vérifier si un programme peut atteindre un état incorrect.
— **Planification automatique**
En intelligence artificielle, certains problèmes de planification sont transformés en formules SAT.

13.15.2 3-SAT

Le problème 3-SAT est une variante de SAT dans laquelle chaque clause contient exactement trois littéraux.
Applications :
— **Configuration de systèmes**
Par exemple dans la configuration automatique de logiciels ou de systèmes industriels.
— **Vérification formelle**
Les solveurs SAT modernes utilisent souvent des formulations en 3-SAT.
— **Optimisation combinatoire**
De nombreux problèmes sont transformés en 3-SAT avant d'être résolus par des solveurs spécialisés.

13.15.3 CLIQUE

Le problème CLIQUE consiste à déterminer si un graphe contient un ensemble de sommets tous connectés entre eux.

Applications :
— **Analyse des réseaux sociaux**
Détection de groupes d'individus fortement connectés.
— **Bioinformatique**
Identification de groupes de protéines interagissant entre elles.
— **Vision par ordinateur**
Correspondance de points dans des images.

13.15.4 Vertex Cover

Le problème Vertex Cover consiste à trouver un ensemble minimal de sommets couvrant toutes les arêtes d'un graphe.

Applications :
— **Surveillance de réseaux**
Placement minimal de capteurs dans un réseau.
— **Sécurité informatique**
Détection des points critiques dans un réseau informatique.
— **Maintenance d'infrastructures**
Choisir un ensemble minimal d'équipements à surveiller.

13.15.5 Cycle Hamiltonien

Le problème du cycle hamiltonien consiste à déterminer si un graphe contient un cycle passant par tous les sommets exactement une fois.

Applications :
— **Planification de tournées**
Organisation de circuits logistiques ou de transport.
— **Conception de circuits électroniques**
Placement de composants sur une carte.
— **Robotique**
Planification de parcours pour robots ou drones.

14 Les problèmes NP-complets

Les problèmes NP-complets occupent une place centrale dans la théorie de la complexité algorithmique.

Ils représentent les problèmes les plus difficiles de la classe NP.

L'étude de ces problèmes permet de comprendre les limites fondamentales des algorithmes efficaces.

14.1 Définition

Un problème est dit **NP-complet** s'il vérifie deux conditions :

1. le problème appartient à la classe NP
2. le problème est NP-difficile

Autrement dit :
— sa solution peut être vérifiée en temps polynomial
— tout problème de NP peut être réduit à ce problème

14.2 Problèmes NP-difficiles

Un problème est **NP-difficile** si tous les problèmes de la classe NP peuvent être réduits polynomialement à lui.

Formellement :

$$\forall A \in NP, \quad A \leq_p B$$

où B est le problème considéré.

Cela signifie que B est au moins aussi difficile que tous les problèmes de NP.

14.3 Relation entre les classes

Les relations entre les principales classes de complexité peuvent être résumées ainsi :

$$P \subseteq NP$$

Les problèmes NP-complets sont un sous-ensemble de NP.

Si un problème NP-complet possède un algorithme polynomial, alors :

$$P = NP$$

14.4 Importance des problèmes NP-complets

Les problèmes NP-complets sont importants car ilsreprésentent un seuil de difficulté en informatique.

Si l'on découvre un algorithme polynomial pour un seul problème NP-complet, alors tous les problèmes de NP peuvent être résolus en temps polynomial.

Cela aurait des conséquences majeures en :

— cryptographie
— optimisation
— intelligence artificielle
— planification
— recherche opérationnelle

14.5 Premier problème NP-complet

Le premier problème NP-complet démontré est le problème SAT (Satisfiability).

Ce résultat est dû au **théorème de Cook-Levin** en 1971.

Ce théorème montre que :

$$SAT \text{ est NP-complet}$$

14.6 Méthode pour démontrer la NP-complétude

Pour prouver qu'un problème B est NP-complet, on suit généralement les étapes suivantes :

1. montrer que $B \in NP$
2. choisir un problème NP-complet connu A
3. construire une réduction polynomiale :

$$A \leq_p B$$

Si ces deux conditions sont satisfaites, alors le problème B est NP-complet.

14.7 Exemples célèbres de problèmes NP-complets

De nombreux problèmes classiques ont été démontrés NP-complets.
— SAT
— 3-SAT
— CLIQUE
— Vertex Cover
— Hamiltonian Cycle
— Traveling Salesman Problem (version décision)
— Subset Sum
— Knapsack

14.8 Chaîne de réductions

La découverte de nouveaux problèmes NP-complets repose souvent sur des chaînes de réductions.

Par exemple :

$$SAT \rightarrow 3SAT \rightarrow CLIQUE \rightarrow VERTEX\ COVER$$

Cela signifie que si l'un de ces problèmes possède un algorithme polynomial, alors tous les autres en possèdent également.

14.9 Le problème du voyageur de commerce

Le **problème du voyageur de commerce** (Traveling Salesman Problem – TSP) est l'un des problèmes les plus célèbres en algorithmique et en optimisation combinatoire.

Définition du problème Étant donné un ensemble de n villes et les distances entre chaque paire de villes, on cherche le plus court circuit qui :
— visite chaque ville exactement une fois,
— revient à la ville de départ.
Formellement, on cherche une permutation des villes minimisant la distance totale parcourue.

Exemple Supposons quatre villes : A, B, C, D.
Un circuit possible est :

$$A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$$

La distance totale est :

$$d(A, B) + d(B, C) + d(C, D) + d(D, A)$$

L'objectif est de trouver le circuit dont la distance est minimale.

Explosion combinatoire Le nombre de circuits possibles augmente très rapidement avec le nombre de villes.
Pour n villes, il existe approximativement :

$$(n-1)!$$

circuits possibles.

Nombre de villes	Nombre de circuits possibles
5	24
10	362 880
15	87 178 291 200
20	2.43×10^{18}

Cette croissance extrêmement rapide illustre le phénomène d'**explosion combinatoire**.

Complexité La méthode naïve consiste à tester toutes les permutations des villes.
Sa complexité est :

$$O(n!)$$

ce qui devient rapidement impossible à calculer pour de grandes valeurs de n.

Nature du problème Le problème du voyageur de commerce appartient à la classe des **problèmes NP-difficiles**. Cela signifie que :
— aucun algorithme polynomial n'est connu pour résoudre ce problème dans le cas général ;
— il est possible de vérifier rapidement une solution proposée ;
— des méthodes heuristiques ou approximatives sont souvent utilisées.

Applications Malgré sa difficulté, ce problème apparaît dans de nombreux domaines :
— optimisation de tournées de livraison ;
— planification de circuits logistiques ;
— conception de circuits électroniques ;
— bioinformatique.
Le TSP constitue donc un exemple classique pour illustrer les limites des algorithmes exacts et l'importance de l'analyse de la complexité.

14.10 Conséquences de P = NP

Si un jour on démontre que :

$$P = NP$$

alors :
— tous les problèmes NP-complets deviennent résolubles efficacement
— de nombreux problèmes d'optimisation seraient faciles à résoudre
— plusieurs systèmes cryptographiques deviendraient vulnérables
Cependant, la majorité des chercheurs pensent que :

$$P \neq NP$$

mais aucune preuve n'a encore été trouvée.

14.11 Approches pour traiter les problèmes NP-complets

Comme ces problèmes sont difficiles, plusieurs stratégies sont utilisées en pratique :
— algorithmes exacts (branch and bound)
— programmation dynamique
— heuristiques
— algorithmes approximatifs
— métaheuristiques (recuit simulé, algorithmes génétiques, recherche taboue)

15 Réduction polynomiale : exemple détaillé

Dans cette section, nous illustrons le principe de la réduction polynomiale à travers un exemple classique :

$$3SAT \leq_p CLIQUE$$

Cette réduction montre que le problème CLIQUE est NP-complet.

15.1 Rappel du problème 3-SAT

Le problème 3-SAT est défini comme suit :

Entrée

Une formule booléenne en forme normale conjonctive dans laquelle chaque clause contient exactement trois littéraux.

Exemple

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

Question

Existe-t-il une affectation des variables qui rend la formule vraie ?

15.2 Rappel du problème CLIQUE

Dans un graphe non orienté $G = (V, E)$, une **clique** est un ensemble de sommets dans lequel chaque paire de sommets est reliée par une arête.

Le problème CLIQUE est défini comme suit.

Entrée

— un graphe G

— un entier k

Question

Existe-t-il une clique de taille k ?

15.3 Idée de la réduction

L'idée est de construire un graphe à partir d'une formule 3-SAT.

Chaque littéral dans une clause correspond à un sommet du graphe.

Les sommets incompatibles ne sont pas reliés.

Une clique de taille égale au nombre de clauses correspond à une sélection cohérente de littéraux satisfaisant toutes les clauses.

15.4 Construction du graphe

Soit une formule 3-SAT contenant m clauses.

La construction du graphe se fait ainsi :

1. créer un sommet pour chaque littéral dans chaque clause
2. connecter deux sommets si :
   — ils appartiennent à des clauses différentes
   — ils ne sont pas contradictoires (ex : x et $\neg x$)

15.5 Exemple

Considérons la formule suivante :

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

On construit alors le graphe :
— 3 sommets pour la première clause
— 3 sommets pour la deuxième clause
On relie les sommets compatibles appartenant à des clauses différentes.

15.6 Interprétation

Dans ce graphe :
— choisir un sommet par clause correspond à choisir un littéral vrai
— une clique correspond à une sélection de littéraux compatibles
Si une clique de taille m existe, alors la formule est satisfiable.

15.7 Correction de la réduction

Nous devons vérifier deux propriétés.
1. Si la formule est satisfiable
Alors on peut choisir un littéral vrai dans chaque clause.
Ces littéraux sont compatibles, donc ils forment une clique.
2. Si une clique de taille m existe
Alors cette clique contient un sommet par clause.
Les littéraux correspondants ne sont pas contradictoires.
Ils définissent donc une affectation satisfaisante.

15.8 Complexité de la transformation

La construction du graphe nécessite :
— création des sommets
— création des arêtes
Ces opérations sont réalisées en temps polynomial.
La réduction est donc polynomiale.

15.9 Conclusion

Nous avons montré que :

$$3SAT \leq_p CLIQUE$$

Comme :
— 3-SAT est NP-complet
— CLIQUE appartient à NP
nous concluons que :

$$CLIQUE \text{ est NP-complet}$$

16 Conclusion

La théorie de la complexité algorithmique permet de mieux comprendre les limites fondamentales du calcul et d'évaluer l'efficacité des algorithmes. Elle fournit un cadre théorique pour analyser les ressources nécessaires à la résolution d'un problème, notamment le temps de calcul et l'espace mémoire, en fonction de la taille des données d'entrée.

L'étude de la complexité met en évidence une distinction essentielle entre les problèmes qui peuvent être résolus efficacement par des algorithmes polynomiaux et ceux dont la résolution semble nécessiter un temps exponentiel. Des exemples classiques, comme le problème du voyageur de commerce, illustrent l'explosion combinatoire qui apparaît lorsque la taille des données augmente.

Ces questions conduisent naturellement à l'un des problèmes les plus importants de l'informatique théorique :

$$P = NP?$$

Autrement dit, tout problème dont une solution peut être vérifiée rapidement peut-il également être résolu rapidement ?

Ce problème, posé dans les années 1970, reste aujourd'hui ouvert et constitue l'un des grands défis scientifiques en informatique théorique. Cette question constitue l'un des problèmes du millénaire en mathématiques et en informatique théorique.

17 Exercices corrigés

Cette section propose quelques exercices des examens des années antérieures permettant de consolider les notions introduites dans ce document. Les corrections détaillées sont également fournies.

Exercice 1 : Questions de cours

Questions

1. Définir la notion de complexité algorithmique.
2. Déduire la définition de la complexité d'un problème.
3. Définir la classe des problèmes NP-complets et donner un exemple.
4. Définir le problème SAT.

Correction

1. La complexité algorithmique mesure les ressources nécessaires à l'exécution d'un algorithme en fonction de la taille de l'entrée n. Les ressources étudiées sont principalement :
   — le temps de calcul (complexité temporelle)
   — l'espace mémoire (complexité spatiale)
2. La complexité d'un problème correspond à la complexité dans le pire des cas du meilleur algorithme capable de résoudre ce problème.
3. Un problème est NP-complet s'il vérifie deux propriétés :
   — il appartient à la classe NP
   — tout problème de NP peut se réduire polynomialement à lui
   Exemple : SAT, 3-SAT, CLIQUE, Vertex Cover.
4. Le problème SAT consiste à déterminer si une formule booléenne peut être rendue vraie en attribuant des valeurs aux variables.

Exercice 2 : Fonction factorielle

1. Écrire une fonction factorielle

```
function factorielle(k)
    f = 1
    for i = 1 to k
        f = f * i
    return f
```

Complexité

La boucle est exécutée k fois.

$$T(k) = O(k)$$

La complexité est donc linéaire.

2. Fonction somme des factorielles

```
function som_fact(L)
    s = 0
    for x in L
        s = s + factorielle(x)
    return s
```

Si la liste contient n éléments et que le maximum vaut m :

$$T(n) = O(n \cdot m)$$

Dans le cas général, la complexité est linéaire par rapport au nombre d'éléments de la liste.

Exercice 3 : Théorème de Cook

Énoncé

Le théorème de Cook-Levin affirme que :

$$SAT \text{ est NP-complet}$$

Idée de la preuve

Toute computation d'une machine de Turing en temps polynomial peut être encodée sous forme de formule booléenne.

Ainsi tout problème de NP peut être transformé en instance du problème SAT.

Cela montre que SAT est NP-difficile.

Comme SAT appartient également à NP, on conclut que SAT est NP-complet.

Exercice 4 : Problème des 8 reines

Le problème consiste à placer n reines sur un échiquier de taille $n \times n$ sans qu'aucune reine ne se menace.

Complexité de l'algorithme EstLibre

L'algorithme vérifie les reines placées dans les lignes précédentes.

Dans le pire des cas, il effectue n comparaisons.

$$T(n) = O(n)$$

La complexité est donc linéaire.

Complexité de l'algorithme PlacerReine

L'algorithme explore un arbre de recherche dans lequel chaque niveau correspond au placement d'une reine.

Chaque reine peut être placée dans au plus n colonnes.

La taille maximale de l'arbre est donc :

$$n!$$

La complexité de l'algorithme est donc :

$$O(n!)$$

Interprétation

Cette complexité illustre une explosion combinatoire. Lorsque n augmente, le nombre de configurations possibles devient extrêmement grand.

Pour cette raison, on utilise souvent des heuristiques ou des techniques de recherche optimisées.

18 Exercices corrigés

Exercice 1 : Questions de cours

Questions

1. Définir la notion de complexité algorithmique.
2. En déduire la définition de la complexité d'un problème.
3. Définir les classes de problèmes P et NP.
4. Définir la notion d'instance d'un problème et donner un exemple.

Correction

1. La complexité algorithmique mesure les ressources nécessaires à l'exécution d'un algorithme en fonction de la taille de l'entrée n. Les ressources étudiées sont principalement :
   — le temps de calcul (complexité temporelle)
   — l'espace mémoire (complexité spatiale)
2. La complexité d'un problème correspond à la complexité du meilleur algorithme capable de résoudre ce problème dans le pire des cas.
3. La classe P est l'ensemble des problèmes pouvant être résolus en temps polynomial.

   $$P = \{L \mid L \text{ est décidable en temps polynomial}\}$$

   La classe NP contient les problèmes dont une solution peut être vérifiée en temps polynomial.

   $$P \subseteq NP$$

4. Une instance d'un problème est un cas particulier du problème.
   Exemple :
   Problème : trouver le maximum d'un tableau.
   Instance :

   $$[3, 7, 2, 9, 5]$$

Exercice 2 : Algorithmes sur les listes

1. Fonction grands(L,x)

```
fonction grands(L,x)
    c = 0
    pour chaque element y dans L
        si y > x alors
            c = c + 1
        fin si
    fin pour
    retourner c
```

Complexité

La boucle parcourt la liste une seule fois.

$$T(n) = O(n)$$

La complexité est linéaire.

2. Fonction petits(L,x)

```
fonction petits(L,x)
    c = 0
    pour chaque element y dans L
        si y < x alors
            c = c + 1
        fin si
    fin pour
    retourner c
```

La complexité est également :

$$O(n)$$

3. Fonction mediane(L)

Une méthode simple consiste à trier la liste.

```
fonction mediane(L)
    trier(L)
    n = longueur(L)
    retourner L[n/2]
```

Si on utilise un tri classique :

$$T(n) = O(n \log n)$$

La complexité de la fonction médiane est donc :

$$O(n \log n)$$

Exercice 3 : Recherche dans un tableau

Algorithme

```
fonction rechercher(T,x,n)
    pour i = 1 a n
        si T[i] = x alors
            retourner vrai
```

```
        fin si
    fin pour
    retourner faux
```

Complexité
Dans le pire cas, on parcourt tout le tableau.

$$T(n) = O(n)$$

Exercice 4 : Complexité de l'algorithme Insertion Sort

L'algorithme de tri par insertion est donné par :

```
for j = 2 to n
    cle = A[j]
    i = j-1
    while i > 0 and A[i] > cle
        A[i+1] = A[i]
        i = i-1
    end while
    A[i+1] = cle
end for
```

Analyse
La boucle externe est exécutée $n-1$ fois.
Dans le pire cas (tableau trié en ordre décroissant), la boucle interne effectue environ :

$$1 + 2 + 3 + ... + (n-1)$$

comparaisons.

$$T(n) = \frac{n(n-1)}{2}$$

Ainsi :

$$T(n) = O(n^2)$$

Conclusion
La complexité du tri par insertion est :
— meilleur cas : $O(n)$
— pire cas : $O(n^2)$

TD : Complexité des algorithmes récursifs

— Savoir établir une relation de récurrence
— Résoudre une récurrence (développement, arbre, théorème maître)
— Analyser la complexité d'algorithmes récursifs

Exercice 1 : Récurrence simple

Soit l'algorithme suivant :

```
def f(n):
    if n == 0:
        return 1
    return f(n-1) + 1
```

1. Donner la relation de récurrence
2. Résoudre la récurrence
3. Donner la complexité

Exercice 2 : Division par 2

```
def g(n):
    if n <= 1:
        return 1
    return g(n//2) + 1
```

1. Donner la récurrence
2. Résoudre
3. Complexité

Exercice 3 : Arbre de récursion

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

1. Représenter l'arbre de récursion
2. Calculer le coût total
3. Déduire la complexité

Exercice 4 : Théorème maître

Résoudre :

1. $T(n) = 2T(n/2) + n$
2. $T(n) = 4T(n/2) + n$
3. $T(n) = 2T(n/2) + n^2$

Exercice 5 : Fibonacci

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

1. Donner la récurrence
2. Montrer que la complexité est exponentielle

Exercice 6 : Algorithme mixte

$$T(n) = 3T(n/2) + n$$

1. Identifier a, b
2. Appliquer le théorème maître
3. Donner la complexité

CORRIGÉ TD

Exercice 1

— Récurrence : $T(n) = T(n-1) + c$
— Développement : $T(n) = T(0) + nc$
— Complexité : $O(n)$

Exercice 2

— Récurrence : $T(n) = T(n/2) + c$
— Développement : $T(n) = T(1) + c \log n$
— Complexité : $O(\log n)$

Exercice 3

Arbre de récursion :
— Niveau 0 : coût $= n$
— Niveau 1 : $2 \times (n/2) = n$
— Niveau 2 : $4 \times (n/4) = n$
Nombre de niveaux $= \log n$

$$T(n) = n \log n$$

Complexité : $O(n \log n)$

18.1 Exercice 4

1. $T(n) = 2T(n/2) + n$

$$T(n) = O(n \log n)$$

2. $T(n) = 4T(n/2) + n$

$$T(n) = O(n^2)$$

3. $T(n) = 2T(n/2) + n^2$

$$T(n) = O(n^2)$$

Exercice 5

— Récurrence : $T(n) = T(n-1) + T(n-2)$
— Arbre exponentiel
— Complexité : $O(2^n)$

Exercice 6

— $a = 3$, $b = 2$
— $n^{\log_2 3} \approx n^{1.58}$
— $f(n) = n$
Donc :

$$T(n) = O(n^{\log_2 3})$$