

# DIU EIL 2020 – Bloc 5

## Retour sur la Complexité

G. Fertin

`guillaume.fertin@univ-nantes.fr`

## Préambule 1/2

### Ce qui est présenté ici

- n'est pas formellement dans le programme NSI de Terminale (!)
- doit être vu comme des éléments de réflexion/d'information supplémentaires
- permet une vision plus large des choses
- et permet d'en savoir plus que vos élèves! (en cas de questions "tordues")
- en gros, fait partie de la "culture générale" en algorithmique

## Préambule 2/2

Ce que nous verrons dans cette partie

1. La **taille des données** d'entrée d'un algorithme, et son importance
2. Le monde algorithmique est binaire: un algorithme est soit **polynomial** soit **exponentiel**
3. Un algorithme résout un problème, mais qu'est-ce qu'un **problème facile** ? un **problème difficile** ?
4. Que fait-on quand un problème est difficile ?

# Sommaire

Taille des Données

Complexité des Algorithmes et Temps d'exécution

Complexité des Problèmes

Conclusion

## Complexité d'un algorithme

C'est l'évaluation des ressources nécessaires à son exécution

- On s'intéresse généralement:
  - à la complexité **en temps** de l'algorithme → rapidité/lenteur
  - à la complexité **en mémoire** de l'algorithme
- Une **tendance** suffit:
  - Notation  $O()$
  - Indépendant de la machine et du langage de programmation
  - Qui reflète assez bien la réalité
  - But: imaginer ce que donne l'algorithme pour de **très grandes données**

## Complexité d'un algorithme

La complexité (en temps/en mémoire) d'un algorithme est une **fonction de la taille des données d'entrée du problème**

## Complexité = fonction de la taille des données

### Intuition

Un même algorithme sur deux tailles différentes:

- Rendu de monnaie pour une somme de  $N=11\text{€}$  vs  $N=547.253\text{€}$  (avez-vous la monnaie sur un billet d'1 million ?)
- Recherche de plus courts chemins dans un graphe à  $n = 27$  sommets vs  $n = 3.000.000$  sommets

### C'est quoi, la "taille des données" ?

Intuitivement:

- c'est la taille de l'instance d'entrée
- c'est ce qu'il faut augmenter si on veut "pousser l'algorithme dans ses retranchements"

## Retour sur la taille des données

### Taille des données (d'entrée d'un algorithme)

- Formellement:

Taille des données = Taille d'un **meilleur codage pour ces données**

- Meilleur codage: codage **informatique** qui prend **le moins de place en mémoire**  
 ⇒ en informatique, penser **codage binaire**!
- Intuitivement (on l'a déjà dit): ce qui va faire augmenter la complexité en temps de l'algorithme



## Meilleur codage 1/2

Au fait, le codage binaire est-il le meilleur possible ?

- Supposons avoir codé un entier  $p$  en base 2  
→ longueur  $\ell_2 = \lceil \log_2 p \rceil$
- Codons-le maintenant en base  $b$  ( $b \geq 2$ ,  $b$  est une constante)  
→ longueur  $\ell_b = \lceil \log_b p \rceil$
- Mais  $\log_2 p = \log_b p \cdot C$ , où  $C = \frac{1}{\log_b 2}$
- ...et  $C$  est une constante (car 2 et  $b$  sont des constantes)
- Conclusion:  $\ell_2$  et  $\ell_b$  sont les mêmes, à une constante multiplicative près
- Remarque: c'est la même idée que la notation  $O(\dots)$

## Meilleur codage 2/2

### Conclusion

Toute base constante  $b \geq 2$  convient. Continuons donc à travailler en base 2!

Seule exception: base  $b = 1$  (codage "bâton", aussi appelé codage **unaire**)

Exemple:  $p = 17$

- Base 2:  $10001_2$  (longueur = 5 bits)
- Base 10:  $17_{10}$  (longueur = 2 chiffres)
- Base 16:  $11_{16}$  (longueur = 2 hexadécimaux)
- Base 1: ||||| (longueur = 17 "bâtons")

## Exercice

### Taille des données

Quelle est la taille correspondant aux données d'entrée suivantes ?

1. **un** entier  $p$  (ex: problème de primalité d'un nombre)
2. **deux** entiers  $p$  et  $q$  (ex: problème du PGCD de deux nombres)
3. un ensemble de  **$n$**  entiers, dont le maximum est  $M$  (ex: problème du tri)
4. un texte de longueur  $n$ , exprimé sur un alphabet  $\Sigma$  de taille  $\sigma$  (ex: problème de la recherche de motifs)
5. un graphe  $G$  à  $n$  sommets et  $m$  arêtes (ex: recherche d'un plus court chemin entre deux sommets de  $G$ )

## Solutions 1/2

### Taille des données

Quelle est la taille correspondant aux données d'entrée suivantes ?

1. **un** entier  $p \rightarrow \log p$

- Pourquoi pas  $\log_2 p$  ?

Réponse: 2 ou autre chose ( $b \geq 2$ ), c'est pareil (voir ci-avant).

Donc on n'indique plus la base du logarithme

- Pourquoi pas  $\lceil \log_2 p \rceil$  ?

Réponse: même idée: si on ne chipote pas pour un écart multiplicatif, on ne chipote pas pour ça non plus!

2. **deux** entiers  $p$  et  $q \rightarrow \log p + \log q$

Mieux! Si on suppose  $p \geq q$ , on se cale sur le plus grand  $\rightarrow 2 \log p$

Voire  $\log p$  puisque la constante multiplicative est "offerte"



## Solutions 2/2

### Taille des données

Quelle est la taille correspondant aux données d'entrée suivantes ?

1. un ensemble de  $n$  entiers, dont le maximum est  $M \rightarrow n \log M$   
(ici, le  $n$  n'est pas "offert", car ce n'est pas une constante!)
4. un texte de longueur  $n$ , exprimé sur un alphabet  $\Sigma$  de taille  $\sigma \rightarrow n \log \sigma$
5. un graphe  $G$  à  $n$  sommets et  $m$  arêtes  $\rightarrow$  plus compliqué:  
c'est quoi le meilleur codage pour un graphe ?
  - Matrice d'adjacence :  $n^2$
  - Liste d'adjacence:  $n + m \log n$  ( $\log n$  pour le codage des sommets)

## Application 1: le tri

**Instance:** une liste  $S$  contenant  $n$  réels

**Sortie:** les  $n$  réels de  $S$  triés par ordre croissant

Taille des données ?

- $n$  valeurs, supposons que le plus grand élément soit  $M$
- Taille des données:  $n \log M$

## Application 1: le tri

**Instance:** une liste  $S$  contenant  $n$  réels

**Sortie:** les  $n$  réels de  $S$  triés par ordre croissant

### Taille des données ?

- Remarque: la valeur de  $M$  n'a aucune importance, c'est l'**ordre relatif** des  $n$  valeurs qui compte
- exemple:  $S_1 = [8, 4, 7, 6, 5]$  et  $S_2 = [8000, 4000, 7000, 6000, 5000]$  se trient de la même façon
- De plus, on admet qu'on peut comparer deux valeurs (même très grandes) en 1 opération
- $\Rightarrow$  la complexité ne dépend pas de  $M$ !

On simplifie, et Taille des données =  $n$

## Application 2: la recherche de motifs

**Instance:** un texte  $T$  de longueur  $n$ , un motif  $M$  de longueur  $m$

**Sortie:** les positions d'apparition de  $M$  dans  $T$

### Taille des données ?

- Alphabet pour exprimer  $T$  et  $M$ :  $\Sigma$ , de taille  $\sigma$
- $T$  de longueur  $n \rightarrow$  meilleur codage  $n \log \sigma$
- $M$  de longueur  $m \rightarrow$  meilleur codage  $m \log \sigma$
- $\sigma$  considéré comme constant (ex: codage ASCII)
- $\Rightarrow$  taille des données =  $n + m$



## Que faut-il conclure de tout ça ?

### Taille des données: en résumé

- une valeur  $p \rightarrow \log p$
- un nombre constant de valeurs, dont la plus grande est  $p_{max}$   
 $\rightarrow \log p_{max}$
- un ensemble de  $n$  valeurs  $\rightarrow n$  (ex: tableaux, listes, piles, files, Arbres Binaires de Recherche...)
- un graphe à  $n$  sommets et  $m$  arêtes  $\rightarrow n + m$  (pourtant, on avait dit  $n^2$  ou  $n + m \log_n$  ??? – sera discuté plus loin)

## Complexité en mémoire d'un algorithme

### Comment mesurer la complexité en mémoire ?

- Complexité en mémoire d'un algorithme = taille prise en mémoire par tout ce qui est stocké quand on exécute l'algorithme:
  - données d'**entrée** du problème
  - toute donnée **supplémentaire** stockée au cours de l'exécution
  - toute donnée de **sortie** (si elle est stockée)

## Application: la recherche de motif

### Retour sur l'algorithme naïf

```
def recherche_naive(T,M):
    n=len(T)
    m=len(M)
    for i in range (n-m):
        j=0
        while j<m and T[i+j]=M[j]:
            j+=1
        if j==M:
            print("Motif trouvé à la position ",i)
```

## Application: la recherche de motif

Retour sur l'algorithme naïf

Complexité en mémoire de l'algo naïf

- $T$  de longueur  $n \rightarrow n \log \sigma$
- $M$  de longueur  $m \rightarrow m \log \sigma$
- variable  $i \rightarrow \log n$
- variable  $j \rightarrow \log m$

$\Rightarrow$  Au total:  $(n + m) \log \sigma + \log n + \log m$

## Complexité en mémoire d'un algorithme

### En résumé

Complexité en mémoire:

- plus simple à calculer que la complexité en temps
- moins crucial de nos jours
- pour la petite (grande?) histoire:

Apollo11 en 1969: 72Ko de mémoire morte + 4Ko de  
mémoire vive

Les temps ont bien changé!

# Sommaire

Taille des Données

Complexité des Algorithmes et Temps d'exécution

Complexité des Problèmes

Conclusion

## $p$ est-il premier ?

Un algorithme possible:

```
booléen premier = VRAI
entier i = 2
```

```
Tant que premier = VRAI et i < p faire
  Si i divise p alors
    premier = FAUX
  FinSi
```

```
  i = i+1
FinTantQue
```

```
return premier
```

## $p$ est-il premier ?

### Analyse de l'algorithme

- Taille des données:  $t = \log p$
- Complexité de l'algorithme:  $O(p)$
- → cette complexité est exponentielle en  $t$  car  $p = 2^t$  (si on choisit la base 2)
- Exemple:  $p = 16.777.216$  et  $t = 24$  (toujours en base 2)
- L'écart entre le codage bâton et le codage binaire est exponentiel!

Remarque: on peut améliorer l'algorithme en remplaçant  $i < p$  par  $i \leq \sqrt{p}$  dans le TantQue

- car si  $p = a \cdot b$ , alors  $\min\{a, b\} \leq \sqrt{p}$
- cela dit, ça ne change rien!  $\sqrt{p} = 2^{\frac{t}{2}}$  reste exponentiel en  $t$



## Quel est le PGCD de $p$ et $q$ ?

L'algorithme d'Euclide (en supposant  $p \geq q$  – sinon, on échange):

```

Tant que q != 0 faire      %%% "!=" veut dire "différent"
    t = q
    q = p mod q
    p = t
FinTantQue

Retourner p

```

- Taille des données:  $t = \log p$
- Nombre d'itérations du Tant Que:  $O(\log p)$
- Donc la complexité est **polynomiale** en  $t$ . Ouf!

## Rendu de monnaie 1/3

### Rappel

- But: Obtenir un certain entier  $N$  en additionnant un minimum d'entiers pris, avec répétition, parmi un ensemble fini  $P = \{p_1, p_2, \dots, p_k\}$
- $N$  : la monnaie à rendre
- $p_1, p_2, \dots, p_k$  : les valeurs faciales des pièces
- Suppositions:
  - $p = 1$  (évite le cas "impossible d'obtenir la valeur  $N$ ")
  - $p_1 < p_2 < \dots < p_k$
  - on a autant de pièces que l'on veut

## Rendu de monnaie 2/3

### Algorithme de Programmation Dynamique

- Idée: remplir une table  $R[S, i]$  pour tous  $0 \leq S \leq N$  et  $0 \leq i \leq k$
- $R[S, i]$  = nombre minimum de pièces dont la somme fait  $S$ , en n'utilisant que les pièces  $p_1, p_2, \dots, p_i$
- Pour le reste, voir les derniers transparents de la Séance 4 du Bloc 5

## Rendu de monnaie 3/3

### Analyse

- Taille des données  $\log N + k \log p_k$
- Complexité de l'algorithme de Prog. Dyn.:  $O(N \cdot k)$
- Problème:  $N$  vs  $\log N$ : cet algorithme est **exponentiel** en la taille des données
- Rem: l'algorithme de Prog. Dyn. est dit **pseudopolynomial** (càd polynomial si on utilisait le codage bâton)

## Alors, tout s'écroule ?

### Flashback

- Quid du problème du sac à dos ?  
Réponse: même problème que le Rendu de monnaie (algorithme pseudopolynomial)
- Quid de la distance d'édition ? (voir Séance 4 du Bloc 5)  
Réponse: OK, car complexité en  $O(n_1 n_2)$  et taille des données  $n_1 \log \sigma + n_2 \log \sigma$  (voire  $n_1 + n_2$ )

## Pour finir sur la taille des données

### Retour sur les graphes

- les algorithmes de graphes que l'on a vus sont polynomiaux en  $n$  et  $m$  (ex:  $O(n + m)$ )
- taille des données:  $n + m \log_n$  en liste d'adjacence
- $\rightarrow$  pas de problème! ( $n$  et  $m$  apparaissent tous deux, et pas au logarithme)
- matrice d'adjacence:  $n^2$ . Convient aussi car  $m \leq n^2$  (donc  $m$  est "capté" par le  $n^2$ )
- on simplifie encore en disant que  $n + m$  est représentatif de la taille des données (et indépendant de la structure!)
- Autre façon de voir: il faut bien stocker les  $n$  sommets et les  $m$  arêtes!

## Pourquoi opposer Polynomial à Exponentiel ?

$n$  = taille des données

### Rappels

- **Exponentielle** : toute fonction en  $O(c^n)$  ( $c$ =constante,  $c > 1$ )
- **Polynomiale** : toute fonction en  $O(n^{c'})$  ( $c'$ =constante)
- Toute fonction exponentielle **domine** toute fonction polynomiale

En ce qui concerne la complexité des algorithmes, on cherche à éviter l'exponentielle

## Pourquoi éviter l'exponentielle ? 1/3

### Exemple illustratif

- $n$ =taille des données
- $f(n)$ =nombre d'opérations=complexité en temps
- une opération prend 1 micro-seconde ( $\mu s$ )=  $10^{-6}s$
- 6 algorithmes, chacun de complexité différente

$n/f(n)$	$n$	$n^2$	$n^3$	$2^n$	$3^n$	$n!$
10	$10\mu s$	$0.1ms$	$1ms$	$1ms$	$59ms$	$3.63s$
20	$20\mu s$	$0.4ms$	$8ms$	$1s$	$58 \text{ min}$	$77094 \text{ ans}$
40	$40\mu s$	$1.6ms$	$64ms$	$12.73 \text{ j}$	$385253 \text{ ans}$	$2.58 \cdot 10^{34} \text{ ans}$
60	$60\mu s$	$3.6ms$	$216ms$	$36533 \text{ ans}$	$1.34 \cdot 10^{15} \text{ ans}$	$2.63 \cdot 10^{68} \text{ ans}$



## Pourquoi éviter l'exponentielle ? 2/3

- Et si on “boostait” ma machine ?
- Et si on utilisait un serveur de calcul ?
- Et si on parallélisait massivement les calculs ?
- Tout cela aide, et permet effectivement d'aller plus vite.
- Mais pour certains complexités, le gain est minime.

Exemple 1: on multiplie par 1.000 la puissance de calcul: une opération prend 1 nano-seconde ( $ns$ ) =  $10^{-9}s$

$n/f(n)$	$n$	$n^2$	$n^3$	$2^n$	$3^n$	$n!$
10	10ns	0.1 $\mu s$	1 $\mu s$	1 $\mu s$	59 $\mu s$	3.63ms
20	20ns	0.4 $\mu s$	8 $\mu s$	1ms	3.48 s	77, 1 ans
40	40ns	1.6 $\mu s$	64 $\mu s$	18.34 h	385, 25 ans	$2.58 \cdot 10^{31}$ ans
60	60ns	3.6 $\mu s$	216 $\mu s$	36, 5 ans	$1.34 \cdot 10^{12}$ ans	$2.63 \cdot 10^{65}$ ans

## Pourquoi éviter l'exponentielle ? 3/3

Exemple 2: on multiplie par **1.000.000** la puissance de calcul: une opération prend 1 **pico**-seconde ( $ps$ ) =  **$10^{-12}$** s

n/f(n)	$n$	$n^2$	$n^3$	$2^n$	$3^n$	$n!$
10	10ps	0.1ns	1ns	1μs	59μs	3.63ms
20	20ps	0.4ns	8ns	1ms	3.48 ms	28.16 j
40	40ps	1.6ns	64ns	1.1 s	140.71 j	$2.58 \cdot 10^{28}$ ans
60	60ps	3.6μs	216ns	13.34 j	$1.34 \cdot 10^9$ ans	$2.63 \cdot 10^{62}$ ans

## En résumé sur cette partie

### Complexité d'un algorithme

- Complexité en mémoire: rarement exponentiel, pas si crucial de nos jours
- Complexité en temps:
  - l'exponentielle guette!
    - si la taille des données est petite et/ou
    - si le problème est "compliqué" à résoudre
  - et on souhaite l'éviter... pour des raisons pratiques (cf. tableaux ci-avant)

# Sommaire

Taille des Données

Complexité des Algorithmes et Temps d'exécution

Complexité des Problèmes

Problèmes de Décision/d'Optimisation

Classes de Complexité

Quelques problèmes NP-complets

Discussion

Conclusion

## Différence entre Algorithme et Problème

### Complexité d'un algorithme

- Jusque là: complexité **des algorithmes**
- Càd: étant donné un algorithme  $A$ , quelle est sa complexité (en temps) ?

### Algorithme vs Problème

- Problème  $\leftrightarrow$  question **générale**
- Algorithme  $\leftrightarrow$  **une** façon de résoudre un problème
- $\Rightarrow$  du coup, que signifie la complexité d'un **problème** ?

## Complexité d'un Problème

La complexité d'un **problème** est la complexité du **meilleur algorithme** qui le résout

Meilleur algorithme = algorithme le plus rapide

## Divers types de Problèmes

### Les problèmes sont de diverses natures – Exemples

- **Problèmes de calcul:**  
 sortie = une ou plusieurs valeur(s) (mais en nombre constant)  
 ex: Racines d'un polynôme du second degré
- **Problèmes d'énumération:**  
 sortie = un ensemble de réponses  
 ex: Recherche de Motif
- **Problèmes d'optimisation:**  
 sortie = une valeur à maximiser ou minimiser  
 ex: Plus court chemin dans un graphe
- **Problèmes de décision:**  
 sortie = oui/non  
 ex:  $p$  est-il premier ?

# Sommaire

Taille des Données

Complexité des Algorithmes et Temps d'exécution

Complexité des Problèmes

Problèmes de Décision/d'Optimisation

Classes de Complexité

Quelques problèmes NP-complets

Discussion

Conclusion



## Problèmes de Décision et d'Optimisation

### Remarque

Dans la suite, nous n'évoquerons que les problèmes **de décision** et **d'optimisation**

### Pourquoi ?

- Problèmes de décision (PbD) “simples” ... à poser
- Problèmes d'optimisation (PbO) très liés aux PbD – on voit ça bientôt
- Les PbD peuvent sembler limités mais
  1. il y en a déjà beaucoup, et
  2. ce qu'on va dire sur eux est applicable aux autres classes de problèmes

## Problèmes de Décision

Réponse = oui/non  $\Rightarrow$  L'énoncé est une question

### Quelques exemples

- Étant donné un entier  $n$ ,  $n$  est-il premier ?
- Étant donnés deux programmes (informatiques)  $\mathcal{P}$  et  $\mathcal{P}'$ ,  $\mathcal{P}$  et  $\mathcal{P}'$  sont-ils équivalents<sup>1</sup> ?
- Étant donné un programme  $\mathcal{P}$ , termine-t-il ?
- Étant donné un graphe  $G$  et un entier  $k$ , peut-on colorier les sommets de  $G$  (sans que deux sommets voisins aient la même couleur<sup>2</sup>), en  $\leq k$  couleurs ?

<sup>1</sup> càd, pour la même instance d'entrée,  $\mathcal{P}$  et  $\mathcal{P}'$  produisent toujours le même résultat

<sup>2</sup> on parle alors de coloration **propre**

## Coloration de Graphes

- Ce problème sera un peu notre fil rouge
- On colorie les sommets de  $G \rightarrow$  on parle de coloration (pas de "coloriage")
- Deux voisins ne peuvent pas avoir la même couleur (on parle de coloration **propre**)
- On cherche à savoir si  $k$  couleurs suffisent

→ Utile pour des allocations de ressources concurrentes

## Coloration de Graphes – Exemple

- Six personnes (nommées  $a, b, c, d, e$  et  $f$ )
- Réservation de créneaux pour une salle de réunion
- Le même jour

Combien de salles sont-elles nécessaires ?

Modélisation par un graphe:

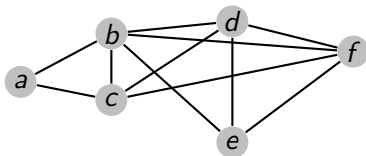
- Sommets: créneaux
- Arêtes: créneaux incompatibles (qui se chevauchent)

## Coloration de Graphes – Exemple

Nom du demandeur	a	b	c
Créneau demandé	8h-10h	9h-14h	9h-12h

Nom du demandeur	d	e	f
Créneau demandé	11h-14h	13h-15h	11h-15h



- Coloration propre en  $k$  couleurs  $\rightarrow k$  salles
- Argument: aucune paire de sommets ayant la même couleur (ex: **vert**) n'est reliée par une arête  
 $\rightarrow$  tout ce qui est **vert** est compatible  
 $\rightarrow$  toutes ces réunions peuvent avoir lieu dans la même salle
- Peut-on colorier ce graphe en 3 couleurs ? en 4 couleurs ? en 5 couleurs ?

## Retour sur les Problèmes de Décision

### Description standard – Exemples

- Formellement, on décrit un PbD sous la forme suivante :
  - NOM**: identifie le problème (en majuscule)
  - Instance**: description des données d'entrée du problème
  - Question**: la question posée

## Problèmes de Décision

### Description standard PbD

#### **PREMIER :**

Instance : un entier  $p$

Question :  $p$  est-il premier ?

#### $k$ -**COL :**

Instance : un graphe  $G$ , un entier  $k$

Question :  $G$  peut-il être colorié de façon propre en  $\leq k$  couleurs ?

## Liens Optimisation/Décision

Optimisation n'est pas décision, mais...

- Tout PbO peut **se ramener** à un PbD
- “Se ramener” signifie:

Existence d'un Algo Polynomial pour PbO

si et seulement si Existence d'un Algo Polynomial pour PbD



## Liens Optimisation/Décision

Illustration sur un exemple

PbO  $\leftrightarrow$  PbD

- Le problème de coloration de graphe!
- En réalité, la vraie question est:

Étant donné un graphe  $G$ , quel est le nombre minimum de couleurs nécessaire pour colorier  $G$  de façon propre ?

- C'est donc (initialement) un problème d'optimisation – appelons-le **MIN-COL**
- Son PbD associé est en réalité **k-COL** (présenté quelques transparents auparavant)

## Liens PbO/PbD

### Passage d'un PbO à son PbD associé

#### MIN-COL :

Instance : un graphe  $G$

Question : quel est le nombre minimum de couleurs nécessaires pour colorier  $G$  de façon propre ?

#### $k$ -COL :

Instance : un graphe  $G$ , un entier  $k$

Question :  $G$  peut-il être proprement colorié en  $\leq k$  couleurs ?

## Liens Optimisation/Décision

Montrons l'équivalence Polynomial(PbO)  $\Leftrightarrow$  Polynomial(PbD)

Polynomial(PbO)  $\Rightarrow$  Polynomial(PbD)

- La résolution de **MIN-COL** donne le nombre  $c_{opt}$  de couleurs
- Concernant **k-COL**:
  - Si  $c_{opt} \leq k \Rightarrow$  réponse **OUI**
  - Si  $c_{opt} > k \Rightarrow$  réponse **NON**
- Intuition: trouver la valeur  $c_{opt}$  est plus dur que savoir si  $c_{opt} \leq k$

## Liens Optimisation/Décision

Montrons l'équivalence Polynomial(PbO)  $\Leftrightarrow$  Polynomial(PbD)

Polynomial(PbD)  $\Rightarrow$  Polynomial(PbO)

- $n$  = nombre de sommets de  $G$
- Remarque: pour tout graphe  $G$ , on sait que  $1 \leq c_{opt} \leq n$
- En effet:
  - $c_{opt} = 1$  pour un graphe sans arête
  - $c_{opt} = n$  si toutes les arêtes possibles sont présentes
- Idée: répondre à **MIN-COL** en faisant des **appels successifs** à  **$k$ -COL**

## Liens Optimisation/Décision

On suppose qu'on a à notre disposition un algorithme  $\text{est-coloriable}(G, k)$ , où  $G$  est un graphe et  $k$  un entier, et qui:

- renvoie VRAI si  $G$  peut être proprement colorié en  $k$  couleurs
- renvoie FAUX sinon
- est polynomial en la taille des données

Algorithme min-coloration( $G$ : graphe)

entier  $i = 0$

booléen  $ok = \text{FAUX}$

Tant que  $ok = \text{FAUX}$  faire

$i = i + 1$

$ok = \text{est-coloriable}(G, i)$

FinTantQue

retourner  $i$

## Liens Optimisation/Décision

Polynomial(PbD)  $\Rightarrow$  Polynomial(PbO)

- Complexité de min-coloration  $\leq n \times$  Complexité de est-coloriable
- Remarque: on peut faire mieux (**dichotomie**)  $\Rightarrow O(\log n)$  appels à est-coloriable

## Liens Optimisation/Décision

### D'une manière générale

- Algo polynomial pour PbO  $\Rightarrow$  Algo polynomial pour PbD: par comparaison entre
  - $k$  = paramètre de PbD
  - $c_{opt}$  = valeur optimale de PbO
- Algo polynomial pour PbD  $\Rightarrow$  Algo polynomial pour PbO: recherche dichotomique pour déterminer  $c_{opt}$

### Remarques

- On parle de **PbD associé à un PbO**
- $\Rightarrow$  A partir de maintenant, le plus souvent, nous observerons des **PbD**

# Sommaire

Taille des Données

Complexité des Algorithmes et Temps d'exécution

Complexité des Problèmes

Problèmes de Décision/d'Optimisation

Classes de Complexité

Quelques problèmes NP-complets

Discussion

Conclusion



# Motivations

## Savoir où on va !

Il y a des problèmes pour lesquels **on ne connaît pas** d'algorithmes efficaces (= temps polynomial)

- Diagnostiquer qu'on se trouve confronté à de tels problèmes
- Ne pas chercher un algorithme efficace, mais plutôt
  - réduire ses exigences ou
  - "contourner" le problème

## La classe P

P =polynomial

En première approche, deux classes de complexité pour les problèmes:

- La **classe P**: tous les problèmes pour lesquels il existe un algorithme de **complexité polynomiale**
- La classe des autres (donc des **exponentiels**, voire pire...)

## La classe P

### Définition P

Un PbD appartient à P s'il existe un algorithme  $A$  et une constante  $c$ , tel que **pour toute instance  $I$**  du PbD :

- $A$  résout le PbD en temps polynomial, càd en  $O(n^c)$
- où  $n$  est la **taille des données**

## Problèmes $\in P$

### Exemples de PbO et PbD $\in P$

- PbO:
  - Problème du Plus Court Chemin (Algorithme de Dijkstra)
  - Problème du PGCD (Algorithme d'Euclide)
- PbD:
  - PREMIER (résultat datant de 2002)  
 Voir par exemple  
[https://fr.wikipedia.org/wiki/Test\\_de\\_primalité\\_AKS](https://fr.wikipedia.org/wiki/Test_de_primalité_AKS)

## La classe P

### En résumé sur P

- Tout  $PbD \in P$  est dit **facile** (ou *tractable*, ou traitable)
- Tout  $PbD \notin P$  est dit **difficile** (ou *intractable*, ou intraitable)

### Exemples de $PbD \notin P$ (et même pire que ça!)

- Terminaison des programmes
- Équivalence des programmes

## La classe P

### Facile ou difficile ?

Problème dans P ou pas dans P ? Pas toujours simple...

- Si le problème  $\in P$ :
  - fournir un algorithme
  - montrer qu'il est correct
  - montrer qu'il est polynomial en la taille des données
- Si le problème  $\notin P$ :
  - montrer qu'**aucun** algorithme polynomial n'existe!
  - en général très compliqué

## P or not P?

Quand on ne sait pas...

- Pour beaucoup de PbD, **on ne sait pas**:
  - **aucun** algo polynomial connu  $\Rightarrow$  tous sont exponentiels...
  - ...mais aucune preuve que le PbD  $\notin$  P !
- Idée: inventer une classe intermédiaire: **NP**

**ATTENTION**

**NP** veut dire Nondeterministic Polynomial

**NP ne veut pas dire NON POLYNOMIAL !**

## Retour à la classe NP

### Définition NP

PbD  $\in$  NP si:

- pour chaque instance positive  $I$  (càd: réponse OUI)
- il existe un **certificat**  $C(I)$  (de sa positivité) vérifiant:
  1. **taille de**  $C(I)$ : **polynomiale** en la taille des données de PbD
  2. **vérification** à partir de  $C(I)$ : en temps **polynomial**



## NP– Intuitivement

### Coloration de graphe

- j'ai un graphe  $G$  et un entier  $k$  (une instance  $I$ )
- je me demande si  $G$  peut être proprement colorié en  $\leq k$  couleurs
- quelqu'un observe par-dessus mon épaule, réfléchit et répond "oui" (instance positive)
- je doute: je lui demande une "preuve" (certificat  $C(I)$ )
- je vérifie, sur la base de sa "preuve", qu'il dit vrai
- si la taille de  $C(I)$  et l'algorithme de vérification sont polynomiaux (en la taille des données), le problème est dans NP

## La classe NP

### Autrement dit

- NP = classe des PbD pour lesquels il est “facile” de **vérifier** qu'une réponse fournie est correcte
- **vérifier**, pas **trouver**
- Les exigences sont donc diminuées → on peut mettre davantage de PbD dans NP
- $\text{PbD} \in P \leftrightarrow \text{solution facile à trouver}$
- $\text{PbD} \in NP \leftrightarrow \text{solution facile à vérifier}$

## La classe NP

### Problème $k$ -COL

- Instance: un graphe  $G$ , un entier  $k$
- Question:  $G$  peut-il être proprement colorié en  $\leq k$  couleurs ?

Montrons que  $k$ -COL  $\in$  NP :

- **Certificat** (pour toute instance  $I$ ) ? dictionnaire sommet  $\leftrightarrow$  couleur pour chacun des  $n$  sommets de  $G$
- Taille du certificat ?  $O(n)$ , donc polynomial
- **Vérification** :
  1. nombre de couleurs  $\leq k$  :  $O(n)$
  2. chaque sommet a exactement une couleur :  $O(n)$
  3. deux sommets voisins dans  $G$  n'ont pas la même couleur :  $O(n^2)$
- Taille et vérification **polynomiales**  $\Rightarrow k$ -COL  $\in$  NP

## La classe NP

### NP, et alors ?

Remarque:  $P \subseteq NP$ :

- trouver est plus dur que vérifier
- si trouver est facile, alors vérifier l'est aussi
- $\Rightarrow P \subseteq NP$

Que faire maintenant avec la classe NP ?

- But: comparer les problèmes  $\in NP$  entre eux
- et surtout, identifier les plus difficiles de NP

$\Rightarrow$  notion de **réduction d'un problème vers un autre**

## Réduction d'un problème vers un autre

### Définition – Réduction

- Soit  $Pb_d$  et  $Pb_a$  deux PbD ( $d$  pour départ,  $a$  pour arrivée)
- Réduction de  $Pb_d$  vers  $Pb_a$  veut dire:
  - partant de **toute instance**  $I_d$  de  $Pb_d$ ...
  - ...on peut créer **en temps polynomial une instance**  $I_a$  de  $Pb_a$
  - de façon que  $I_d$  et  $I_a$  ont **la même réponse** (OUI ou NON)
- On écrira  $Pb_d \geq_P Pb_a$  ( $Pb_d$  se réduit en temps polynomial vers  $Pb_a$ )

## Réduction

### Idée principale

But: comparer les PbD de NP entre eux

En effet, si  $Pb_d \geq_P Pb_a$ , alors:

- Si  $Pb_a$  est dans P, alors  $Pb_d$  l'est aussi (car on peut résoudre  $Pb_d$  en “passant par”  $Pb_a$ )
- $\Rightarrow$   **$Pb_a$**  est au moins aussi difficile que  **$Pb_d$**

## Les problèmes NP-complets

NP + réduction  $\Rightarrow$  les problèmes **les plus difficiles de NP**

Nouvelle classe: NP-complet

### Définition NP-complet

Un PbD est NP-complet si :

1. il est dans NP
2. **chaque problème de NP peut se réduire vers lui**

Abus de langage: on dit “est NP-complet” au lieu de  $\in$  NP-complet

## Astuce

- Vous avez bien dit “**chaque** problème NP peut se réduire...” !!!
- Question: comment fait-on ?
- Réponse: ci-dessous

Pour montrer qu'un PbD  $Pb$  est NP-complet, il “suffit” de montrer:

1.  $Pb \in NP$
2. il existe **un** PbD NP-complet  $Pb_1$  tel que  $Pb_1 \geq_P Pb$

Dit autrement:

- **une seule** réduction suffit
- mais il faut réduire depuis un problème NP-complet
- (ça se démontre assez simplement – mais pas ici!)



## Les problèmes NP-complets

### En conclusion

- Un PbD NP-complet est un problème NP **au moins aussi difficile que tout autre** problème NP
- Les PbD NP-complets sont les problèmes **les plus difficiles de NP**

# Sommaire

Taille des Données

Complexité des Algorithmes et Temps d'exécution

Complexité des Problèmes

Problèmes de Décision/d'Optimisation

Classes de Complexité

Quelques problèmes NP-complets

Discussion

Conclusion

## L'œuf et la poule

Montrer qu'un PbD est NP-complet implique de réduire un problème NP-complet vers lui... mais il faut bien commencer quelque part!

### Remarque

Il n'est a priori pas évident qu'il existe au moins un problème NP-complet

### Theorem (Cook 1971)

*Le problème **SAT** est NP-complet*

- **SAT** est un problème de logique booléenne
- Démontré via la machine de Turing

## L'œuf et la poule

- Depuis 1971, c'est les soldes: les réductions pleuvent!
- → les PbD NP-complets sont **nombreux** !
- Par exemple:
  - $k$ -**COL** est NP-complet pour tout  $k \geq 3$
  - le problème du Sac à Dos est NP-complet
  - le problème de Rendu de Monnaie est NP-complet

## L'œuf et la poule

Pour aller plus loin:

[https://fr.wikipedia.org/wiki/Liste\\_de\\_problèmes\\_NP-complets](https://fr.wikipedia.org/wiki/Liste_de_problèmes_NP-complets)

et aussi la “bible” dans le domaine:

“Computers and Intractability: A Guide to the Theory of NP-Completeness” de Garey et Johnson

Paru en 1979

Qui liste plus de 300 problèmes NP-complets

## Attention à NP-complet!

### Ce que veut dire NP-complet

Un PbD est NP-complet veut dire:

- qu'il existe **au moins une** instance "compliquée à résoudre"
- mais pas forcément toutes !!!
- des **cas particuliers** (potentiellement nombreux) peuvent être polynomiaux
- ...on en parle un peu plus tard

# Sommaire

Taille des Données

Complexité des Algorithmes et Temps d'exécution

Complexité des Problèmes

Problèmes de Décision/d'Optimisation

Classes de Complexité

Quelques problèmes NP-complets

Discussion

Conclusion

## Pour revenir à la classe NP-complet

A quoi ça peut bien servir ?

1. Si **un seul** PbD NP-complet est polynomial  $\Rightarrow$  **tous les problèmes** NP aussi !
2. Inversement, si **un seul** PbD NP est intraitable  $\Rightarrow$  **tous les problèmes** NP-complets aussi !



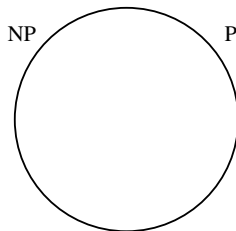
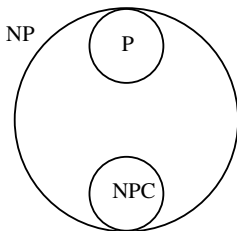
## Quelques propriétés des NP-complets

### Problèmes NP-complets: où en est-on ?

- Problèmes tous très difficiles à résoudre
- **Aucun algorithme polynomial** n'a été trouvé pour ces problèmes
- L'impossibilité de trouver des algorithmes polynomiaux n'a pas été prouvée non plus !
- L'existence ou non d'algorithmes polynomiaux pour les problèmes NP-complets est **l'un des plus grands problèmes encore ouverts en informatique**

## Relations entre P et NP ...

- Les problèmes NP-complets sont les problèmes les plus difficiles de la classe NP
- $P \subseteq NP$
- mais qu'est-ce qui est correct ? le schéma de gauche, ou celui de droite ?



## La grande question

$P = NP?$

- Recherches innombrables sur le sujet depuis des dizaines d'années
- Fait partie des 7 problèmes du millénaire du Clay Mathematics Institute  
<http://www.claymath.org/millennium-problems>  
 Remarque: 1 million de dollars par problème résolu
- Les implications sont multiples et réelles ! Exemple: transactions bancaires cryptées sur le web (codage RSA)
- On ne sait toujours pas:
  - si les  $PbD \in NP$  sont tous polynomiaux...
  - ou si les problèmes NP-complets sont exponentiels

## La grande question

$P = NP?$

L'opinion généralement partagée est que  $P \subset NP$ , c'àd les problèmes NP-complets **ne sont pas** polynomiaux

## Concrètement

### Que faire face à un problème inconnu ?

On observe notre problème  $Pb$

- soit on pense que le problème  $Pb$  est **facile**  
 ⇒ on cherche un algorithme **correct** et polynomial (avec le meilleur temps possible!) qui le résout
- soit on pense que le problème  $Pb$  est **difficile**  
 ⇒ on cherche à montrer qu'il est NP-complet, càd
  - toute solution proposée peut être polynomialement **vérifiable** (appartenance à NP)
  - prendre un problème NP-complet et le **réduire** polynomialement à  $Pb$

## Que faire face à un problème Polynomial ?

Si  $Pb$  est polynomial

- Trouver le “meilleur” algorithme
- S'assurer qu'il est **correct**!
- Meilleur = le plus **rapide**

Exemple: le problème du Tri

## Que faire face à un problème NP-complet?

Si  $P_b$  est NP-complet

- Premier constat: ne pas s'acharner à trouver un algorithme exact et rapide qui fonctionne sur toutes les instances
- Baisser ses exigences:
  - (a) soit sur la **rapidité** d'exécution  
Je veux la réponse exacte, je suis prêt.e à attendre (si la taille est petite, ça ira)
  - (b) soit sur l'**exactitude** de la réponse  
Je veux une réponse rapide, tant pis si elle n'est pas tout à fait exacte
  - (c) soit sur l'**ensemble des instances** autorisées  
Je peux avoir un algorithme rapide et exact si mes données d'entrée sont "gentilles"

## Que faire face à un problème NP-complet?

### Discussion

- La très grande majorité des PbO/PbD “intéressants” sont NP-complets
- Le problème de décision  $\mathcal{P}$  est NP-complet signifie que:
  1. Il existe **au moins une instance** de  $\mathcal{P}$  qui est difficile à résoudre
  2. Il **n'existe pas d'algorithme polynomial** (en temps) pour résoudre  $\mathcal{P}$  (sauf si  $P = NP$ )
  3.  $\mathcal{P}$  est au moins aussi difficile que les problèmes **SAT**, **MIN-COL**, **SAC A DOS**, **RENDU DE MONNAIE**, ...
- En l'état actuel des connaissances:
  - résultat **optimal**  $\Rightarrow$  temps d'exécution **déraisonnable**
  - temps d'exécution **raisonnable**  $\Rightarrow$  résultat **pas optimal**



## Que faire face à un problème NP-complet?

### Retour sur le cas (c)

Je peux avoir un algorithme rapide et exact si mes données d'entrée sont "gentilles"

- NP-complet signifie qu'**au moins une instance** est "difficile" ...
- ...mais **pas forcément toutes** !
- Pour **certaines instances**, le problème (pourtant NP-complet) pourrait être résolu en temps polynomial
- Exemples:
  - **MIN-COL** limité aux graphes de degré maximum 2
  - **MIN-COL** limité aux arbres

## MIN-COL limité aux graphes de degré maximum 2

**MIN-COL** limité aux graphes de degré maximum 2 :

Instance : un graphe  $G$  **de degré maximum 2**

Question : quel est le nombre minimum de couleurs nécessaires pour colorier  $G$  de façon propre ?

### Exercice

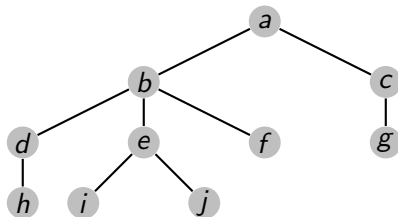
- Si  $G$  est connexe et de degré max. 2, à quoi ressemble  $G$  ?
- Si  $G$  n'est pas (forcément) connexe, à quoi ressemble-t-il ?
- Montrer que le problème **MIN-COL** limité aux graphes de degré max. 2 est dans P

## MIN-COL limité aux arbres

### MIN-COL :

Instance : un **arbre**  $G$

Question : quel est le nombre minimum de couleurs nécessaires pour colorier  $G$  de façon propre ?



### Exercice

Montrer que le problème **MIN-COL** limité aux arbres est dans P

## Que faire face à un problème NP-complet?

### Retour sur le cas (b)

Je veux une réponse rapide, tant pis si elle n'est pas tout à fait exacte

- S'applique surtout aux PbO
- temps d'exécution exigé: **polynomial**
- Une possibilité: Algorithmes d'**approximation**:
  - algorithme polynomial
  - garantissant un résultat  $\leq r \cdot c_{opt}$  (maximisation) ou  $\geq r \cdot c_{opt}$  (minimisation)
  - pour **toutes les instances**
  - $r$  est appelé le **ratio d'approximation**

## Retour sur le problème du **SAC A DOS**

### Théorème

Pour tout  $\varepsilon > 0$ , il existe un algorithme de  $(1 - \varepsilon)$ -approximation pour **SAC A DOS**, et dont la complexité en temps est en  $O(N^3 \cdot \frac{1}{\varepsilon})$

Rappels:

- $N$  = nombre d'éléments
- Ici, ratio  $r = 1 - \varepsilon$ , où  $\varepsilon$  est aussi petit que l'on veut (avec  $\varepsilon > 0$ , tout de même...)

Discussion:

- c'est le cas le plus favorable (compromis qualité du résultat/temps d'exécution)
- utilise l'algorithme de Prog. Dynamique (sur une instance "divisée") – tout n'est pas perdu!

## Approximation pour **SAC A DOS**

### Les grandes idées

Partant d'une instance  $I$ :

- **diviser** les valeurs par un même facteur  $X$  pour diminuer l'échelle sur ces valeurs
- **arrondir**  $\rightarrow$  instance  $I'$
- utiliser l'**algo de Prog. Dyn.** sur  $I'$
- **remonter la solution** (exacte) pour  $I'$  vers une solution (approchée) pour  $I$

# Sommaire

Taille des Données

Complexité des Algorithmes et Temps d'exécution

Complexité des Problèmes

Conclusion

## Résumé en quelques transparents

### Taille des Données

- En général, ne pose pas de problèmes
- Rester attentif aux problèmes ayant en entrée:
  - un nombre constant de valeurs (ex: **PREMIER, PGCD**)
  - des valeurs “cruciales” pour résoudre le problème (ex: **SAC A DOS, RENDU DE MONNAIE**)



## Résumé en quelques transparents

### Complexité des Problèmes

- Tout une théorie existe (seulement effleurée ici)
- Permet d'estimer la difficulté des problèmes (P vs NP-complet)
- Si le problème est NP-complet, on adapte sa stratégie de résolution
- Le catalogue d'algorithmes est immense
  - heuristiques
  - algorithmes d'approximation
  - algorithmes probabilistes
  - algorithmes de complexité paramétrée
  - programmation linéaire
  - programmation par contraintes
  - apprentissage automatique
  - etc.