Structures de données et algorithmes fondamentaux

01 – Complexité algorithmique

Anthony Labarre

20 octobre 2015

UNIVERSITÉ PARIS-EST MARNE-LA-VALLÉE

iut Marne-la-Vallée

Informations pratiques

- Mes coordonnées :
  - Anthony Labarre
  - E-mail : Anthony.Labarre@univ-mlv.fr
  - Bureau : 4B069 (bâtiment Copernic, 4ème étage)
  - Page du cours : `http://igm.univ-mlv.fr/~alabarre/`
  - Volume (prévu) :
    - cours : 2h/semaine (5 séances)
    - TD : 2h/semaine (2 groupes, 7 séances) ;
    - TP : 2h/semaine (2 groupes, 10 séances) ;
- Les supports de cours seront disponibles à l'adresse ci-dessus ;
- Evaluation en continu :
  - séances de travaux dirigés ;
  - séances de travaux pratiques (à rendre) ;
  - examen : à déterminer ;

L'algorithmique

- L'objet de ce cours est **l'algorithmique** ;

L'algorithmique

- L'objet de ce cours est **l'algorithmique** ;
- L'algorithmique est l'étude des **algorithmes** ;

L'algorithmique

- L'objet de ce cours est **l'algorithmique** ;
- L'algorithmique est l'étude des **algorithmes** ;
- Un **algorithme** est une méthode permettant de résoudre un problème donné en un temps fini ;

L'algorithmique

- L'objet de ce cours est **l'algorithmique** ;
- L'algorithmique est l'étude des **algorithmes** ;
- Un **algorithme** est une méthode permettant de résoudre un problème donné en un temps fini ;

$\boxed{\text{problème}}$

L'algorithmique

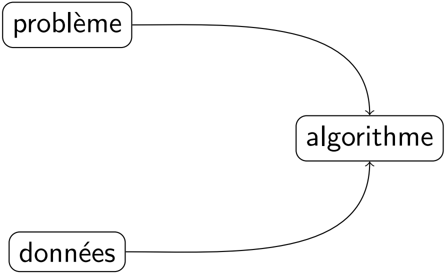
- L'objet de ce cours est **l'algorithmique** ;
- L'algorithmique est l'étude des **algorithmes** ;
- Un **algorithme** est une méthode permettant de résoudre un problème donné en un temps fini ;

problème

données

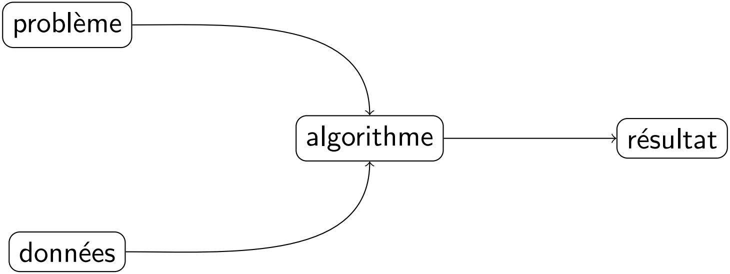
L'algorithmique

- L'objet de ce cours est **l'algorithmique** ;
- L'algorithmique est l'étude des **algorithmes** ;
- Un **algorithme** est une méthode permettant de résoudre un problème donné en un temps fini ;



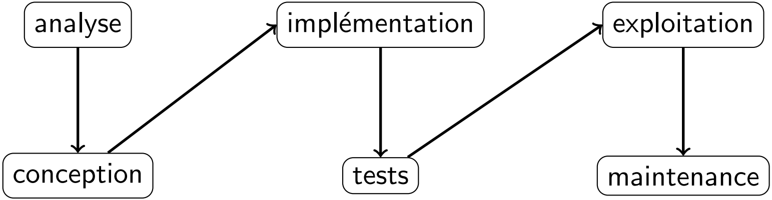
L'algorithmique

- L'objet de ce cours est **l'algorithmique** ;
- L'algorithmique est l'étude des **algorithmes** ;
- Un **algorithme** est une méthode permettant de résoudre un problème donné en un temps fini ;



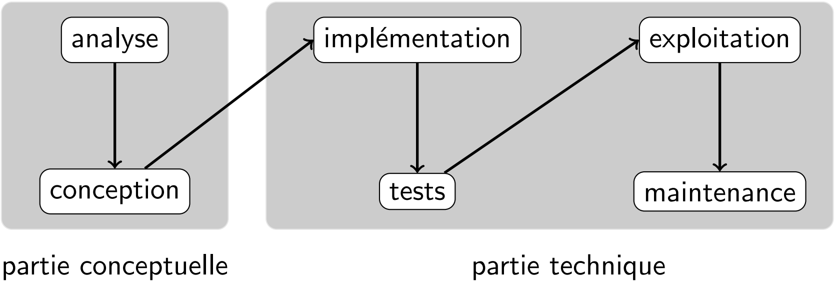
Développement de logiciels

- **Un algorithme n'est pas un programme !**
  - l'algorithme décrit une méthode qui sera ensuite implémentée dans un langage de programmation ;
- Le développement d'un programme se divise en plusieurs phases :



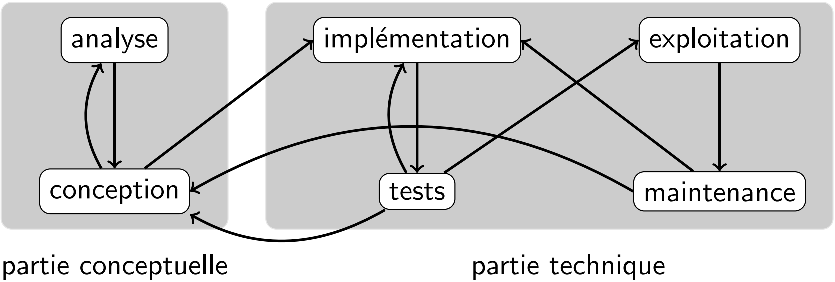
Développement de logiciels

- **Un algorithme n'est pas un programme !**
  - l'algorithme décrit une méthode qui sera ensuite implémentée dans un langage de programmation ;
- Le développement d'un programme se divise en plusieurs phases :



Développement de logiciels

- **Un algorithme n'est pas un programme !**
  - l'algorithme décrit une méthode qui sera ensuite implémentée dans un langage de programmation ;
- Le développement d'un programme se divise en plusieurs phases :



- L'algorithmique se situe au niveau conceptuel ;
- En pratique, les choses sont souvent plus compliquées ...

L'algorithmique

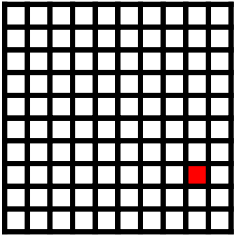
- En algorithmique, on veut expliquer (à un ordinateur) comment résoudre efficacement un problème ;

L'algorithmique

- En algorithmique, on veut expliquer (à un ordinateur) comment résoudre efficacement un problème ;
- Remarquons que certains problèmes sont évidents pour un humain, moins pour un ordinateur ;

L'algorithmique

- En algorithmique, on veut expliquer (à un ordinateur) comment résoudre efficacement un problème ;
- Remarquons que certains problèmes sont évidents pour un humain, moins pour un ordinateur ;
- Exemple simple : où est la case rouge ?



Méthodologie

- Il n'existe pas d'algorithme pour créer des algorithmes ;

Méthodologie

- Il n'existe pas d'algorithme pour créer des algorithmes ;
- Cela dit, il existe quelques principes généraux que l'on peut suivre ;

Méthodologie

- Il n'existe pas d'algorithme pour créer des algorithmes ;
- Cela dit, il existe quelques principes généraux que l'on peut suivre ;
- Les TDs et TPs vous serviront à acquérir ces principes ;

Méthodologie

- Il n'existe pas d'algorithme pour créer des algorithmes ;
- Cela dit, il existe quelques principes généraux que l'on peut suivre ;
- Les TDs et TPs vous serviront à acquérir ces principes ;
- Les tâches à accomplir étant parfois complexes, il est important de les décomposer et d'établir un plan à suivre ;

Méthodologie : sous-problèmes

- Principe : on découpe une tâche complexe en tâches plus simples ;

Méthodologie : sous-problèmes

- Principe : on découpe une tâche complexe en tâches plus simples ;
- Exemple : créer un réseau social (Facebook, etc.) :

Méthodologie : sous-problèmes

- Principe : on découpe une tâche complexe en tâches plus simples ;
- Exemple : créer un réseau social (Facebook, etc.) :
  - stocker et sauvegarder les données ;

Méthodologie : sous-problèmes

- Principe : on découpe une tâche complexe en tâches plus simples ;
- Exemple : créer un réseau social (Facebook, etc.) :
  - stocker et sauvegarder les données ;
  - représenter les utilisateurs et leurs attributs ;

Méthodologie : sous-problèmes

- Principe : on découpe une tâche complexe en tâches plus simples ;
- Exemple : créer un réseau social (Facebook, etc.) :
  - stocker et sauvegarder les données ;
  - représenter les utilisateurs et leurs attributs ;
  - interface ;

Méthodologie : sous-problèmes

- Principe : on découpe une tâche complexe en tâches plus simples ;
- Exemple : créer un réseau social (Facebook, etc.) :
  - stocker et sauvegarder les données ;
  - représenter les utilisateurs et leurs attributs ;
  - interface ;
  - gestion de la vie privée ;

Méthodologie : sous-problèmes

- Principe : on découpe une tâche complexe en tâches plus simples ;
- Exemple : créer un réseau social (Facebook, etc.) :
  - stocker et sauvegarder les données ;
  - représenter les utilisateurs et leurs attributs ;
  - interface ;
  - gestion de la vie privée ;

  ...

Du raisonnement à l'algorithme puis au code

- Exemple de tâche : décider si une liste L est triée ;

Du raisonnement à l'algorithme puis au code

- Exemple de tâche : décider si une liste L est triée ;
  - L est triée si tous ses éléments sont dans l'ordre croissant ;

Du raisonnement à l'algorithme puis au code

- Exemple de tâche : décider si une liste L est triée ;
  - L est triée si tous ses éléments sont dans l'ordre croissant ;
  - Plus formellement :

$$L \text{ triée } \Leftrightarrow \forall\ 0 \leq i < |L| - 1 : L[i] \leq L[i+1]$$

Du raisonnement à l'algorithme puis au code

- Exemple de tâche : décider si une liste L est triée ;
  - L est triée si tous ses éléments sont dans l'ordre croissant ;
  - Plus formellement :

$$L \text{ triée } \Leftrightarrow \forall\, 0 \leq i < |L| - 1 : L[i] \leq L[i+1]$$

- On en déduit l'algorithme suivant : supposer la liste triée au départ, et chercher une contradiction ;

Du raisonnement à l'algorithme puis au code

- Exemple de tâche : décider si une liste L est triée ;
  - L est triée si tous ses éléments sont dans l'ordre croissant ;
  - Plus formellement :

$$L \text{ triée } \Leftrightarrow \forall\ 0 \leq i < |L| - 1 : L[i] \leq L[i+1]$$

- On en déduit l'algorithme suivant : supposer la liste triée au départ, et chercher une contradiction ;
- Ceci donne par exemple en code :

```
def est_triee(L):
    for i in range(len(L) - 1):
        if L[i] > L[i + 1]:
            return False
    return True
```

Sujets traités dans ce cours

- Un algorithme doit résoudre de manière **efficace** le problème donné, **quelles que soient les données à traiter** ;

Sujets traités dans ce cours

- Un algorithme doit résoudre de manière **efficace** le problème donné, **quelles que soient les données à traiter** ;
- On s'intéresse principalement à deux aspects d'un algorithme donné :
  1. sa correction : résout-il bien le problème donné ?
  2. son efficacité : en combien de temps et avec quelles ressources ?

Sujets traités dans ce cours

- Un algorithme doit résoudre de manière **efficace** le problème donné, **quelles que soient les données à traiter** ;
- On s'intéresse principalement à deux aspects d'un algorithme donné :
  1. sa correction : résout-il bien le problème donné ?
  2. son efficacité : en combien de temps et avec quelles ressources ?
- On verra :
  - comment concevoir des algorithmes ;
  - comment démontrer leur correction et analyser leur efficacité ;
  - diverses structures de données et des algorithmes classiques pour les manipuler ;

Sujets traités dans ce cours

- Un algorithme doit résoudre de manière **efficace** le problème donné, **quelles que soient les données à traiter** ;
- On s'intéresse principalement à deux aspects d'un algorithme donné :
  1. sa correction : résout-il bien le problème donné ?
  2. son efficacité : en combien de temps et avec quelles ressources ?
- On verra :
  - comment concevoir des algorithmes ;
  - comment démontrer leur correction et analyser leur efficacité ;
  - diverses structures de données et des algorithmes classiques pour les manipuler ;
- Les implémentations seront en Python, mais ce cours n'est pas un cours de Python ;
  - pour les bases, voir le cours de programmation ;

Algorithmique, pas Python

- Comme il s'agit d'un cours d'algorithmique, le but est de vous montrer des choses plus générales que Python ;
- Certains algorithmes seront donc présentés de manière à ce que vous soyiez capables de les implémenter dans n'importe quel langage "sans trop de problèmes" ;
- On va donc tenter de ne pas trop avoir recours à des astuces "propres à Python" ;
  - par exemple : une fonction plutôt que == pour une comparaison ;
- Moins esthétique, mais plus général ;
- La connaissance du fonctionnement de ces algorithmes permet :
  - de les évaluer ;
  - de les améliorer ;

Motivation

- Rappel : on n'exige pas seulement d'un algorithme qu'il résolve un problème ;

Motivation

- Rappel : on n'exige pas seulement d'un algorithme qu'il résolve un problème ;
- On veut également qu'il soit **efficace**, c'est-à-dire :

Motivation

- Rappel : on n'exige pas seulement d'un algorithme qu'il résolve un problème ;
- On veut également qu'il soit **efficace**, c'est-à-dire :
  - rapide (en termes de temps d'exécution) ;
  - économe en ressources (espace de stockage, mémoire utilisée) ;

Motivation

- Rappel : on n'exige pas seulement d'un algorithme qu'il résolve un problème ;
- On veut également qu'il soit **efficace**, c'est-à-dire :
  - rapide (en termes de temps d'exécution) ;
  - économe en ressources (espace de stockage, mémoire utilisée) ;
- On a donc besoin d'outils qui nous permettront d'évaluer la qualité **théorique** des algorithmes proposés ;

Motivation

- Rappel : on n'exige pas seulement d'un algorithme qu'il résolve un problème ;
- On veut également qu'il soit **efficace**, c'est-à-dire :
  - rapide (en termes de temps d'exécution) ;
  - économe en ressources (espace de stockage, mémoire utilisée) ;
- On a donc besoin d'outils qui nous permettront d'évaluer la qualité **théorique** des algorithmes proposés ;
- On se focalisera dans un premier temps sur le temps d'exécution ;

Exemple : deux méthodes de recherche dans les listes

Voici deux méthodes vérifiant si un élément appartient à une liste :

Version courte

```python
def f(T, x):
    if x in T:
        return True
    return False
```

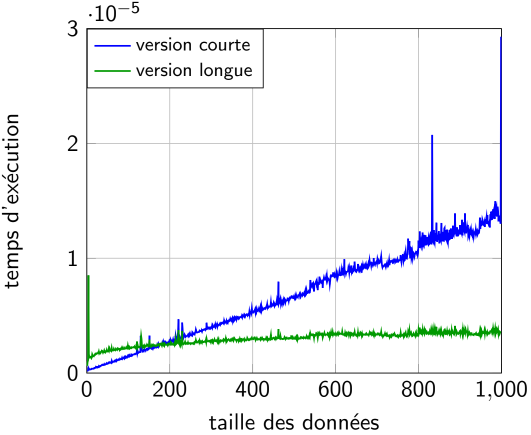
Version longue

```python
def g(T, x, a=0, b=0):
    if a > b:
        return False
    p = (a + b) // 2
    if T[p] == x:
        return True
    if T[p] > x:
        return g(T, x, a, p-1)
    return g(T, x, p+1, b)
```

Laquelle choisir ?

Exemple : deux méthodes de recherche dans les listes

La mesure du temps d'exécution en pratique nous aide :



... mais ne nous dit pas pourquoi l'une est meilleure que l'autre.

Théorie de la complexité

- La **théorie de la complexité** (algorithmique) vise à répondre à ces besoins ;
- Elle permet :
  1. de classer les problèmes selon leur difficulté ;
  2. de classer les algorithmes selon leur efficacité ;
  3. de comparer les algorithmes résolvant un problème donné afin de faire un choix éclairé <u>sans</u> devoir les implémenter ;

Evaluation de la rapidité d'un algorithme

Evaluation de la rapidité d'un algorithme

- On ne mesure pas la durée en heures, minutes, secondes, ... :
  1. cela impliquerait d'implémenter les algorithmes qu'on veut comparer ;
  2. de plus, ces mesures ne seraient pas pertinentes car le même algorithme sera plus rapide sur une machine plus puissante ;

Evaluation de la rapidité d'un algorithme

- On ne mesure pas la durée en heures, minutes, secondes, ... :
  1. cela impliquerait d'implémenter les algorithmes qu'on veut comparer ;
  2. de plus, ces mesures ne seraient pas pertinentes car le même algorithme sera plus rapide sur une machine plus puissante ;
- Au lieu de ça, on utilise des **unités de temps abstraites** proportionnelles au nombre d'opérations effectuées ;

Evaluation de la rapidité d'un algorithme

- On ne mesure pas la durée en heures, minutes, secondes, ... :
    1. cela impliquerait d'implémenter les algorithmes qu'on veut comparer ;
    2. de plus, ces mesures ne seraient pas pertinentes car le même algorithme sera plus rapide sur une machine plus puissante ;
- Au lieu de ça, on utilise des **unités de temps abstraites** proportionnelles au nombre d'opérations effectuées ;
- Au besoin, on pourra ensuite adapter ces quantités en fonction de la machine sur laquelle l'algorithme s'exécute ;

Calcul du temps d'exécution

- Chaque **instruction basique** consomme une unité de temps ; (affectation d'une variable, comparaison, $+$, $-$, $*$, $/$, ...)

Calcul du temps d'exécution

- Chaque **instruction basique** consomme une unité de temps ; (affectation d'une variable, comparaison, $+$, $-$, $*$, $/$, ...)
- Chaque **itération** d'une boucle rajoute le nombre d'unités de temps consommées dans le corps de cette boucle ;

Calcul du temps d'exécution

- Chaque **instruction basique** consomme une unité de temps ; (affectation d'une variable, comparaison, $+$, $-$, $*$, $/$, ...)
- Chaque **itération** d'une boucle rajoute le nombre d'unités de temps consommées dans le corps de cette boucle ;
- Chaque **appel de fonction** rajoute le nombre d'unités de temps consommées dans cette fonction ;

Calcul du temps d'exécution

- Chaque **instruction basique** consomme une unité de temps ; (affectation d'une variable, comparaison, $+$, $-$, $*$, $/$, ...)
- Chaque **itération** d'une boucle rajoute le nombre d'unités de temps consommées dans le corps de cette boucle ;
- Chaque **appel de fonction** rajoute le nombre d'unités de temps consommées dans cette fonction ;
- Pour avoir le nombre d'opérations effectuées par l'algorithme, on additionne le tout ;

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- L'algorithme suivant calcule $n! = n * (n-1) * (n-2) * \cdots * 2 * 1$ (avec $0! = 1$) :

Exemple (factorielle de n)

```python
def factorielle(n):
    fact = 1
    i = 2
    while i <= n:
        fact = fact * i
        i = i + 1
    return fact
```

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- L'algorithme suivant calcule $n! = n * (n-1) * (n-2) * \cdots * 2 * 1$ (avec $0! = 1$) :

Exemple (factorielle de n)

```
def factorielle(n):
    fact = 1
    i = 2
    while i <= n:
        fact = fact * i
        i = i + 1
    return fact
```

initialisation : 1

initialisation : 1

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- L'algorithme suivant calcule $n! = n * (n-1) * (n-2) * \cdots * 2 * 1$ (avec $0! = 1$) :

Exemple (factorielle de n)

```
def factorielle(n):
    fact = 1
    i = 2
    while i <= n:
        fact = fact * i
        i = i + 1
    return fact
```

initialisation : 1

initialisation : 1

itérations : au plus $n-1$

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- L'algorithme suivant calcule $n! = n * (n-1) * (n-2) * \cdots * 2 * 1$ (avec $0! = 1$) :

Exemple (factorielle de n)

`def factorielle(n):`	
`    fact = 1`	initialisation : 1
`    i = 2`	initialisation : 1
`    while i <= n:`	itérations : au plus $n-1$
`        fact = fact * i`	multiplication + affectation : 2
`        i = i + 1`	addition + affectation : 2
`    return fact`	

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- L'algorithme suivant calcule $n! = n * (n-1) * (n-2) * \cdots * 2 * 1$ (avec $0! = 1$) :

Exemple (factorielle de n)

```
def factorielle(n):
    fact = 1                          initialisation : 1
    i = 2                             initialisation : 1
    while i <= n:                     itérations : au plus n − 1
        fact = fact * i               multiplication + affectation : 2
        i = i + 1                     addition + affectation : 2
    return fact                       renvoi d'une valeur : 1
```

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- L'algorithme suivant calcule $n! = n * (n-1) * (n-2) * \cdots * 2 * 1$ (avec $0! = 1$) :

Exemple (factorielle de n)

```
def factorielle(n):
    fact = 1                          initialisation : 1
    i = 2                             initialisation : 1
    while i <= n:                     itérations : au plus n − 1
        fact = fact * i               multiplication + affectation : 2
        i = i + 1                     addition + affectation : 2
    return fact                       renvoi d'une valeur : 1
```

- On a aussi un test à chaque itération ;
- Nombre total d'opérations :

$$1 + 1 + (n-1) * 5 + 1 + 1 = 5n - 1$$

Remarques

- Ce genre de calcul n'est pas tout à fait exact ;

Remarques

- Ce genre de calcul n'est pas tout à fait exact ;
- En effet, on ne sait pas toujours combien de fois exactement on va effectuer une boucle ;

Remarques

- Ce genre de calcul n'est pas tout à fait exact ;
- En effet, on ne sait pas toujours combien de fois exactement on va effectuer une boucle ;
- De même, lors d'un branchement conditionnel, le nombre de comparaisons effectuées n'est pas toujours le même ;
  - par exemple, si a est faux dans le test $a \wedge b$, alors évaluer b est inutile puisque $a \wedge b$ sera toujours faux ;

Remarques

- Ce genre de calcul n'est pas tout à fait exact ;
- En effet, on ne sait pas toujours combien de fois exactement on va effectuer une boucle ;
- De même, lors d'un branchement conditionnel, le nombre de comparaisons effectuées n'est pas toujours le même ;
  - par exemple, si a est faux dans le test $a \wedge b$, alors évaluer b est inutile puisque $a \wedge b$ sera toujours faux ;
- Heureusement, on dispose d'outils plus simples à manipuler et plus adaptés ;

Mesure du temps d'exécution en Python

- Une façon de mesurer le temps d'exécution de bouts de code en Python est de simuler un chronomètre :

Mesure du temps d'exécution en Python

- Une façon de mesurer le temps d'exécution de bouts de code en Python est de simuler un chronomètre :
  - on le déclenche juste avant le début du bout de code ;

Mesure du temps d'exécution en Python

- Une façon de mesurer le temps d'exécution de bouts de code en Python est de simuler un chronomètre :
  - on le déclenche juste avant le début du bout de code ;
  - on l'arrête juste après la fin du bout de code ;

Mesure du temps d'exécution en Python

- Une façon de mesurer le temps d'exécution de bouts de code en Python est de simuler un chronomètre :
  - on le déclenche juste avant le début du bout de code ;
  - on l'arrête juste après la fin du bout de code ;
  - le temps écoulé entre les deux pressions est la durée qui nous intéresse ;

Mesure du temps d'exécution en Python

- Une façon de mesurer le temps d'exécution de bouts de code en Python est de simuler un chronomètre :
  - on le déclenche juste avant le début du bout de code ;
  - on l'arrête juste après la fin du bout de code ;
  - le temps écoulé entre les deux pressions est la durée qui nous intéresse ;
- On peut y parvenir grâce au module `time` ;

Mesure du temps d'exécution en Python

- Une façon de mesurer le temps d'exécution de bouts de code en Python est de simuler un chronomètre :
  - on le déclenche juste avant le début du bout de code ;
  - on l'arrête juste après la fin du bout de code ;
  - le temps écoulé entre les deux pressions est la durée qui nous intéresse ;
- On peut y parvenir grâce au module `time` ;

Exemple (utilisation du chronomètre)

```
from time import time

debut = time()        # on enclenche le chronomètre
# morceau de code
fin = time()              # on arrête le chronomètre
print('Temps écoulé:', fin - debut, 'secondes')
```

Décimales

Exemple

```
>>> from time import time
>>> debut = time()
>>> fin = time()
>>> print(fin - debut, 'secondes')
2.4835598468780518 secondes
```

- On n'a pas toujours envie de connaître toutes les décimales ;
- Si par exemple on veut n'en garder que deux, on peut :
  1. multiplier la mesure par 100 ;
  2. tronquer la partie décimale ;
  3. diviser le résultat par 100.
- En Python : `int(duree * 100) / 100` ;

Complexité algorithmique

- La **complexité** d'un algorithme est une mesure de sa performance **asymptoptique** dans le **pire cas** ;

Complexité algorithmique

- La **complexité** d'un algorithme est une mesure de sa performance **asymptoptique** dans le **pire cas** ;
- Que signifie **asymptotique** ?

Complexité algorithmique

- La **complexité** d'un algorithme est une mesure de sa performance **asymptoptique** dans le **pire cas** ;
- Que signifie **asymptotique** ?
  $\hookrightarrow$ on s'intéresse à des données très grandes ;

Complexité algorithmique

- La **complexité** d'un algorithme est une mesure de sa performance **asymptoptique** dans le **pire cas** ;
- Que signifie **asymptotique** ?
  $\hookrightarrow$ on s'intéresse à des données très grandes ;
- Que signifie "dans le pire cas" ?

Complexité algorithmique

- La **complexité** d'un algorithme est une mesure de sa performance **asymptoptique** dans le **pire cas** ;
- Que signifie **asymptotique** ?
  $\hookrightarrow$ on s'intéresse à des données très grandes ;
- Que signifie "dans le pire cas" ?
  $\hookrightarrow$ on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à résoudre ;

Complexité algorithmique

- La **complexité** d'un algorithme est une mesure de sa performance **asymptoptique** dans le **pire cas** ;
- Que signifie **asymptotique** ?
  $\hookrightarrow$ on s'intéresse à des données très grandes ;
  - pourquoi ?
- Que signifie "dans le pire cas" ?
  $\hookrightarrow$ on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à résoudre ;

Complexité algorithmique

- La **complexité** d'un algorithme est une mesure de sa performance **asymptoptique** dans le **pire cas** ;
- Que signifie **asymptotique** ?
  $\hookrightarrow$ on s'intéresse à des données très grandes ;
  - pourquoi ?
    $\hookrightarrow$ les petites valeurs ne sont pas assez informatives ;
- Que signifie "dans le pire cas" ?
  $\hookrightarrow$ on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à résoudre ;

Complexité algorithmique

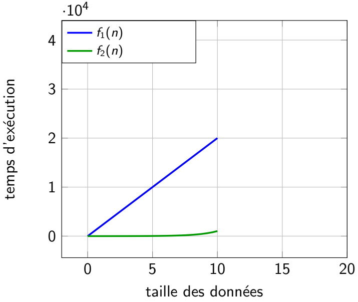
- La **complexité** d'un algorithme est une mesure de sa performance **asymptoptique** dans le **pire cas** ;
- Que signifie **asymptotique** ?
  $\hookrightarrow$ on s'intéresse à des données très grandes ;
  - pourquoi ?
    $\hookrightarrow$ les petites valeurs ne sont pas assez informatives ;
- Que signifie "dans le pire cas" ?
  $\hookrightarrow$ on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à résoudre ;
  - pourquoi ?

Complexité algorithmique

- La **complexité** d'un algorithme est une mesure de sa performance **asymptoptique** dans le **pire cas** ;
- Que signifie **asymptotique** ?
  $\hookrightarrow$ on s'intéresse à des données très grandes ;
  - pourquoi ?
    $\hookrightarrow$ les petites valeurs ne sont pas assez informatives ;
- Que signifie "dans le pire cas" ?
  $\hookrightarrow$ on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à résoudre ;
  - pourquoi ?
    $\hookrightarrow$ on veut être sûr que l'algorithme ne prendra jamais plus de temps que ce qu'on a estimé ;

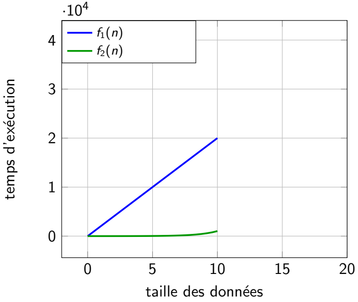
Intuition : comportement asymptotique

- Soit
  1. un problème à résoudre sur des données de taille n, et
  2. deux algorithmes résolvant ce problème en un temps $f_1(n)$ et $f_2(n)$, respectivement ;
- Quel algorithme préférez-vous ?



Intuition : comportement asymptotique

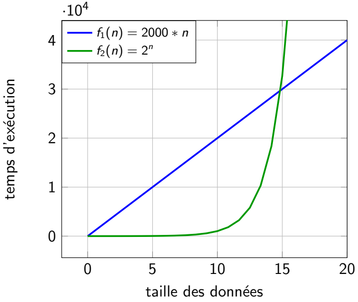
- Soit
  1. un problème à résoudre sur des données de taille n, et
  2. deux algorithmes résolvant ce problème en un temps $f_1(n)$ et $f_2(n)$, respectivement ;
- Quel algorithme préférez-vous ?



- La courbe verte semble correspondre à un algorithme beaucoup plus efficace ...

Intuition : comportement asymptotique

- Soit
  1. un problème à résoudre sur des données de taille n, et
  2. deux algorithmes résolvant ce problème en un temps $f_1(n)$ et $f_2(n)$, respectivement ;
- Quel algorithme préférez-vous ?



- La courbe verte semble correspondre à un algorithme beaucoup plus efficace ...
- ... mais seulement pour de très petites valeurs !

La notation $O(\cdot)$: motivation

- Les calculs à effectuer pour évaluer le temps d'exécution d'un algorithme peuvent parfois être longs et pénibles ;

La notation $O(\cdot)$: motivation

- Les calculs à effectuer pour évaluer le temps d'exécution d'un algorithme peuvent parfois être longs et pénibles ;
- De plus, le degré de précision qu'ils requièrent est souvent inutile ;

La notation $O(\cdot)$: motivation

- Les calculs à effectuer pour évaluer le temps d'exécution d'un algorithme peuvent parfois être longs et pénibles ;
- De plus, le degré de précision qu'ils requièrent est souvent inutile ;
- On aura donc recours à une **approximation** de ce temps de calcul, représentée par la notation $O(\cdot)$;

Notation

- Soit n la taille des données à traiter ; on dit qu'une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

$$\exists\ n_0 \in \mathbb{N}, \exists\ c \in \mathbb{R}, \forall\ n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

Notation

- Soit n la taille des données à traiter ; on dit qu'une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

$$\exists\ n_0 \in \mathbb{N}, \exists\ c \in \mathbb{R}, \forall\ n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

- Autrement dit : $f(n)$ est en $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f(\cdot)$ est toujours dominée par la fonction $g(\cdot)$, à une constante multiplicative fixée près ;

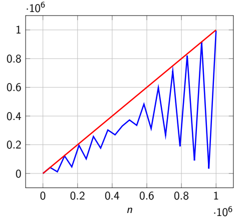
Notation

- Soit n la taille des données à traiter ; on dit qu'une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

$$\exists\ n_0 \in \mathbb{N}, \exists\ c \in \mathbb{R}, \forall\ n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

- Autrement dit : $f(n)$ est en $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f(\cdot)$ est toujours dominée par la fonction $g(\cdot)$, à une constante multiplicative fixée près ;

Exemple (quelques cas où $f(n) = O(g(n))$)



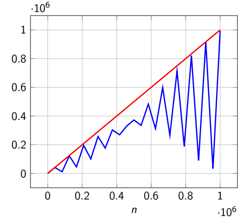
Notation

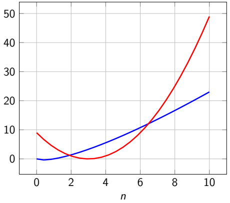
- Soit n la taille des données à traiter ; on dit qu'une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

$$\exists\ n_0 \in \mathbb{N}, \exists\ c \in \mathbb{R}, \forall\ n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

- Autrement dit : $f(n)$ est en $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f(\cdot)$ est toujours dominée par la fonction $g(\cdot)$, à une constante multiplicative fixée près ;

Exemple (quelques cas où $f(n) = O(g(n))$)





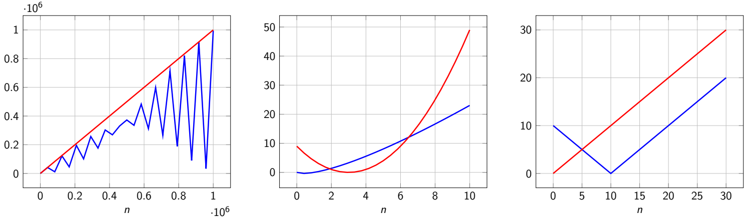
Notation

- Soit n la taille des données à traiter ; on dit qu'une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

$$\exists\ n_0 \in \mathbb{N}, \exists\ c \in \mathbb{R}, \forall\ n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

- Autrement dit : $f(n)$ est en $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f(\cdot)$ est toujours dominée par la fonction $g(\cdot)$, à une constante multiplicative fixée près ;

Exemple (quelques cas où $f(n) = O(g(n))$)



Exemples d'utilisation de $O(\cdot)$

- Prouvons que la fonction $f_1(n) = 5n + 37$ est en $O(n)$:

Exemples d'utilisation de $O(\cdot)$

- Prouvons que la fonction $f_1(n) = 5n + 37$ est en $O(n)$:
  1. but : trouver une constante $c \in \mathbb{R}$ et un seuil $n_0 \in \mathbb{N}$ à partir duquel $|f_1(n)| \leq c|n|$.

Exemples d'utilisation de $O(\cdot)$

- Prouvons que la fonction $f_1(n) = 5n + 37$ est en $O(n)$:
  1. but : trouver une constante $c \in \mathbb{R}$ et un seuil $n_0 \in \mathbb{N}$ à partir duquel $|f_1(n)| \leq c|n|$.
  2. on remarque que $|5n + 37| \leq |6n|$ si $n \geq 37$:

$$\begin{aligned} |5 * 37 + 37| &\leq 6 * |37|\,; \\ |5 * 38 + 37| &\leq 6 * |38|\,; \\ &\vdots \end{aligned}$$

Exemples d'utilisation de $O(\cdot)$

- Prouvons que la fonction $f_1(n) = 5n + 37$ est en $O(n)$:
  1. but : trouver une constante $c \in \mathbb{R}$ et un seuil $n_0 \in \mathbb{N}$ à partir duquel $|f_1(n)| \leq c|n|$.
  2. on remarque que $|5n + 37| \leq |6n|$ si $n \geq 37$:

$$\begin{aligned} &|5 * 37 + 37| \leq 6 * |37|\,; \\ &|5 * 38 + 37| \leq 6 * |38|\,; \\ &\qquad\vdots \end{aligned}$$

  3. on en déduit donc que $c = 6$ fonctionne à partir du seuil $n_0 = 37$, et on a fini.

Exemples d'utilisation de $O(\cdot)$

- Prouvons que la fonction $f_1(n) = 5n + 37$ est en $O(n)$:
  1. but : trouver une constante $c \in \mathbb{R}$ et un seuil $n_0 \in \mathbb{N}$ à partir duquel $|f_1(n)| \leq c|n|$.
  2. on remarque que $|5n + 37| \leq |6n|$ si $n \geq 37$:

     $$\begin{array}{l} |5 * 37 + 37| \leq 6 * |37|\,; \\ |5 * 38 + 37| \leq 6 * |38|\,; \\ \qquad\vdots \end{array}$$

  3. on en déduit donc que $c = 6$ fonctionne à partir du seuil $n_0 = 37$, et on a fini.
- Remarque : on ne demande pas d'optimisation (le plus petit c ou n_0 qui fonctionne), juste de donner des valeurs qui fonctionnent ;
  - $c = 10$ et $n_0 = 8$ est donc aussi acceptable ;

Exemples d'utilisation de $O(\cdot)$

- Prouvons que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $O(n^2)$:

Exemples d'utilisation de $O(\cdot)$

- Prouvons que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $O(n^2)$:
  1. cherchons d'abord la constante c ; $c = 6$ ne peut pas marcher, essayons donc $c = 7$;

Exemples d'utilisation de $O(\cdot)$

- Prouvons que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $O(n^2)$:
  1. cherchons d'abord la constante c ; $c = 6$ ne peut pas marcher, essayons donc $c = 7$;
  2. on doit alors trouver un seuil $n_0 \in \mathbb{N}$ à partir duquel :

$$|6n^2 + 2n - 8| \leq 7|n^2| \qquad \forall\ n \geq n_0;$$

Exemples d'utilisation de $O(\cdot)$

- Prouvons que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $O(n^2)$:
  1. cherchons d'abord la constante c ; $c = 6$ ne peut pas marcher, essayons donc $c = 7$;
  2. on doit alors trouver un seuil $n_0 \in \mathbb{N}$ à partir duquel :

$$|6n^2 + 2n - 8| \leq 7|n^2| \qquad \forall\ n \geq n_0;$$

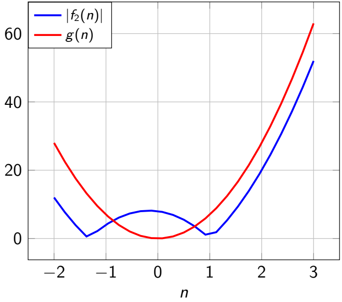
  3. un simple calcul nous donne $n_1 = -\frac{4}{3}$ et $n_2 = 1$ comme racines de l'équation $6n^2 + 2n - 8 = 0$;

Exemples d'utilisation de $O(\cdot)$

- Prouvons que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $O(n^2)$:
  1. cherchons d'abord la constante c ; $c = 6$ ne peut pas marcher, essayons donc $c = 7$;
  2. on doit alors trouver un seuil $n_0 \in \mathbb{N}$ à partir duquel :

  $$|6n^2 + 2n - 8| \leq 7|n^2| \qquad \forall\ n \geq n_0;$$

  3. un simple calcul nous donne $n_1 = -\frac{4}{3}$ et $n_2 = 1$ comme racines de l'équation $6n^2 + 2n - 8 = 0$;
  4. en conclusion, $c = 7$ et $n_0 = 1$ nous donnent le résultat voulu ;



Règles de calcul : simplifications

- On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :

Règles de calcul : simplifications

- On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :
  1. on oublie les constantes multiplicatives (elles valent 1) ;

Règles de calcul : simplifications

- On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :
  1. on oublie les constantes multiplicatives (elles valent 1) ;
  2. on annule les constantes additives ;

Règles de calcul : simplifications

- On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :
  1. on oublie les constantes multiplicatives (elles valent 1) ;
  2. on annule les constantes additives ;
  3. on ne retient que les termes dominants ;

Règles de calcul : simplifications

- On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :
  1. on oublie les constantes multiplicatives (elles valent 1) ;
  2. on annule les constantes additives ;
  3. on ne retient que les termes dominants ;

Exemple (simplifications)

Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

Règles de calcul : simplifications

- On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :
  1. on oublie les constantes multiplicatives (elles valent 1) ;
  2. on annule les constantes additives ;
  3. on ne retient que les termes dominants ;

Exemple (simplifications)

Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$

Règles de calcul : simplifications

- On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :
  1. on oublie les constantes multiplicatives (elles valent 1) ;
  2. on annule les constantes additives ;
  3. on ne retient que les termes dominants ;

Exemple (simplifications)

Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
2. on annule les constantes additives : $n^3 - n^2 + n + 0$

Règles de calcul : simplifications

- On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :
  1. on oublie les constantes multiplicatives (elles valent 1) ;
  2. on annule les constantes additives ;
  3. on ne retient que les termes dominants ;

Exemple (simplifications)

Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
2. on annule les constantes additives : $n^3 - n^2 + n + 0$
3. on garde le terme de plus haut degré : $n^3 + 0$

Règles de calcul : simplifications

- On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :
  1. on oublie les constantes multiplicatives (elles valent 1) ;
  2. on annule les constantes additives ;
  3. on ne retient que les termes dominants ;

Exemple (simplifications)

Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
2. on annule les constantes additives : $n^3 - n^2 + n + 0$
3. on garde le terme de plus haut degré : $n^3 + 0$

et on a donc $g(n) = O(n^3)$.

Justification des simplifications

- Les processeurs actuels effectuent plusieurs milliards d'opérations à la seconde ;

Justification des simplifications

- Les processeurs actuels effectuent plusieurs milliards d'opérations à la seconde ;
  1. qu'une affectation requière 2 ou 4 unités de temps ne change donc pas grand-chose ;

Justification des simplifications

- Les processeurs actuels effectuent plusieurs milliards d'opérations à la seconde ;
  1. qu'une affectation requière 2 ou 4 unités de temps ne change donc pas grand-chose ;
  2. un nombre constant d'instructions est donc aussi négligeable par rapport à la croissance de la taille des données ;

Justification des simplifications

- Les processeurs actuels effectuent plusieurs milliards d'opérations à la seconde ;
  1. qu'une affectation requière 2 ou 4 unités de temps ne change donc pas grand-chose ;
  2. un nombre constant d'instructions est donc aussi négligeable par rapport à la croissance de la taille des données ;
  3. pour de grandes valeurs de n, le terme de plus haut degré l'emportera ;

Justification des simplifications

- Les processeurs actuels effectuent plusieurs milliards d'opérations à la seconde ;
  1. qu'une affectation requière 2 ou 4 unités de temps ne change donc pas grand-chose ;
     $\hookrightarrow$ d'où le remplacement des constantes par des 1 pour les multiplications
  2. un nombre constant d'instructions est donc aussi négligeable par rapport à la croissance de la taille des données ;
  3. pour de grandes valeurs de n, le terme de plus haut degré l'emportera ;

Justification des simplifications

- Les processeurs actuels effectuent plusieurs milliards d'opérations à la seconde ;
  1. qu'une affectation requière 2 ou 4 unités de temps ne change donc pas grand-chose ;
     $\hookrightarrow$ d'où le remplacement des constantes par des 1 pour les multiplications
  2. un nombre constant d'instructions est donc aussi négligeable par rapport à la croissance de la taille des données ;
     $\hookrightarrow$ d'où l'annulation des constantes additives
  3. pour de grandes valeurs de n, le terme de plus haut degré l'emportera ;

Justification des simplifications

- Les processeurs actuels effectuent plusieurs milliards d'opérations à la seconde ;
  1. qu'une affectation requière 2 ou 4 unités de temps ne change donc pas grand-chose ;
     $\hookrightarrow$ d'où le remplacement des constantes par des 1 pour les multiplications
  2. un nombre constant d'instructions est donc aussi négligeable par rapport à la croissance de la taille des données ;
     $\hookrightarrow$ d'où l'annulation des constantes additives
  3. pour de grandes valeurs de n, le terme de plus haut degré l'emportera ;
     $\hookrightarrow$ d'où l'annulation des termes inférieurs

Justification des simplifications

- Les processeurs actuels effectuent plusieurs milliards d'opérations à la seconde ;
  1. qu'une affectation requière 2 ou 4 unités de temps ne change donc pas grand-chose ;
     $\hookrightarrow$ d'où le remplacement des constantes par des 1 pour les multiplications
  2. un nombre constant d'instructions est donc aussi négligeable par rapport à la croissance de la taille des données ;
     $\hookrightarrow$ d'où l'annulation des constantes additives
  3. pour de grandes valeurs de n, le terme de plus haut degré l'emportera ;
     $\hookrightarrow$ d'où l'annulation des termes inférieurs
- On préfére donc avoir une **idée** du temps d'exécution de l'algorithme plutôt qu'une expression plus précise mais inutilement compliquée ;

Règles de calculs : combinaison des complexités

- Les instructions de base prennent un temps constant, noté $\mathbf{O(1)}$;
- On additionne les complexités d'opérations en séquence :

$$O(f_1(n)) + O(f_2(n)) = O(f_1(n) + f_2(n))$$

- Même chose pour les branchements conditionnels :

Exemple

```
if <condition>:            O(g(n))
    # instructions (1)     O(f1(n))
else:
    # instructions (2)     O(f2(n))
```

$$\left.\begin{array}{l} O(g(n)) \\ O(f_1(n)) \\ \\ O(f_2(n)) \end{array}\right\} = O(g(n) + f_1(n) + f_2(n))$$

Règles de calculs : combinaison des complexités

- Dans les boucles, on multiplie la complexité du corps de la boucle par le nombre d'itérations ;
- La complexité d'une boucle `while` se calcule comme suit :

Exemple

```
# en supposant qu'on a m itérations
while <condition>:
    # instructions
```

$$\left.\begin{array}{l} O(g(n)) \\ O(f(n)) \end{array}\right\} = O(m * (g(n) + f(n)))$$

Calcul de la complexité d'un algorithme

- Pour calculer la complexité d'un algorithme :

Calcul de la complexité d'un algorithme

- Pour calculer la complexité d'un algorithme :
  1. on calcule la complexité de chaque "partie" de l'algorithme ;

Calcul de la complexité d'un algorithme

- Pour calculer la complexité d'un algorithme :
  1. on calcule la complexité de chaque "partie" de l'algorithme ;
  2. on combine ces complexités conformément aux règles qu'on vient de voir ;

Calcul de la complexité d'un algorithme

- Pour calculer la complexité d'un algorithme :
  1. on calcule la complexité de chaque “partie” de l'algorithme ;
  2. on combine ces complexités conformément aux règles qu'on vient de voir ;
  3. on simplifie le résultat grâce aux règles de simplifications qu'on a vues ;
     - élimination des constantes, et
     - conservation du (des) terme(s) dominant(s)

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- Reprenons le calcul de la factorielle, qui nécessitait $5n - 1$ opérations :

Exemple (factorielle de n)

```
def factorielle(n):
    fact = 1
    i = 2
    while i <= n:
        fact = fact * i
        i = i + 1
    return fact
```

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- Reprenons le calcul de la factorielle, qui nécessitait $5n - 1$ opérations :

Exemple (factorielle de n)

```
def factorielle(n):
    fact = 1
    i = 2
    while i <= n:
        fact = fact * i
        i = i + 1
    return fact
```

initialisation : $O(1)$

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- Reprenons le calcul de la factorielle, qui nécessitait $5n - 1$ opérations :

Exemple (factorielle de n)

```
def factorielle(n):
    fact = 1
    i = 2
    while i <= n:
        fact = fact * i
        i = i + 1
    return fact
```

initialisation : $O(1)$

initialisation : $O(1)$

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- Reprenons le calcul de la factorielle, qui nécessitait $5n - 1$ opérations :

Exemple (factorielle de n)

```
def factorielle(n):
    fact = 1                 initialisation :      O(1)
    i = 2                    initialisation :      O(1)
    while i <= n:            n − 1 itérations :    O(n)
        fact = fact * i
        i = i + 1
    return fact
```

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- Reprenons le calcul de la factorielle, qui nécessitait $5n - 1$ opérations :

Exemple (factorielle de n)

```
def factorielle(n):
    fact = 1                  initialisation :     O(1)
    i = 2                     initialisation :     O(1)
    while i <= n:             n − 1 itérations :   O(n)
        fact = fact * i       multiplication :     O(1)
        i = i + 1
    return fact
```

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- Reprenons le calcul de la factorielle, qui nécessitait $5n - 1$ opérations :

Exemple (factorielle de n)

```
def factorielle(n):
    fact = 1                  initialisation :      O(1)
    i = 2                     initialisation :      O(1)
    while i <= n:             n − 1 itérations :    O(n)
        fact = fact * i       multiplication :      O(1)
        i = i + 1             incrémentation :      O(1)
    return fact
```

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- Reprenons le calcul de la factorielle, qui nécessitait $5n - 1$ opérations :

Exemple (factorielle de n)

```
def factorielle(n):
    fact = 1                  initialisation :     O(1)
    i = 2                     initialisation :     O(1)
    while i <= n:             n − 1 itérations :   O(n)
        fact = fact * i       multiplication :     O(1)
        i = i + 1             incrémentation :     O(1)
    return fact               renvoi :             O(1)
```

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- Reprenons le calcul de la factorielle, qui nécessitait $5n - 1$ opérations :

Exemple (factorielle de n)

`def factorielle(n):`		
`    fact = 1`	initialisation :	$O(1)$
`    i = 2`	initialisation :	$O(1)$
`    while i <= n:`	$n-1$ itérations :	$O(n)$
`        fact = fact * i`	multiplication :	$O(1)$
`        i = i + 1`	incrémentation :	$O(1)$
`    return fact`	renvoi :	$O(1)$

- Complexité de la procédure :

$$O(1) + O(n) * O(1) + O(1) = O(n)$$

Equivalence de fonctions en termes de $O(\cdot)$

- $f(n) = O(g(n))$ n'implique pas $g(n) = O(f(n))$:
  - contre-exemple : $5n + 43 = O(n^2)$, mais $n^2 \neq O(n)$;
- $f(n) \neq O(g(n))$ n'implique pas $g(n) \neq O(f(n))$:
  - contre-exemple : $18n^3 - 35n \neq O(n)$, mais $n = O(n^3)$;

Equivalence de fonctions en termes de $O(\cdot)$

- $f(n) = O(g(n))$ n'implique pas $g(n) = O(f(n))$:
  - contre-exemple : $5n + 43 = O(n^2)$, mais $n^2 \neq O(n)$;
- $f(n) \neq O(g(n))$ n'implique pas $g(n) \neq O(f(n))$:
  - contre-exemple : $18n^3 - 35n \neq O(n)$, mais $n = O(n^3)$;
- On dit que deux fonctions $f(n)$ et $g(n)$ sont **équivalentes** si

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

Equivalence de fonctions en termes de $O(\cdot)$

- $f(n) = O(g(n))$ n'implique pas $g(n) = O(f(n))$:
  - contre-exemple : $5n + 43 = O(n^2)$, mais $n^2 \neq O(n)$;
- $f(n) \neq O(g(n))$ n'implique pas $g(n) \neq O(f(n))$:
  - contre-exemple : $18n^3 - 35n \neq O(n)$, mais $n = O(n^3)$;
- On dit que deux fonctions $f(n)$ et $g(n)$ sont **équivalentes** si

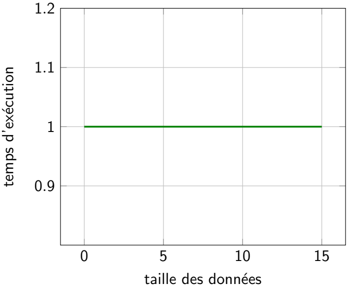
$$f(n) = O(g(n)) \text{ et } g(n) = O(f(n))$$

- D'un point de vue algorithmique, trouver un nouvel algorithme de même complexité pour un problème donné ne présente donc pas beaucoup d'intérêt ;

Quelques classes de complexité

- On peut donc “ranger” les fonctions équivalentes dans la même **classe** ;
- Voici quelques classes fréquentes de complexité (par ordre croissant en termes de $O(\cdot)$) :

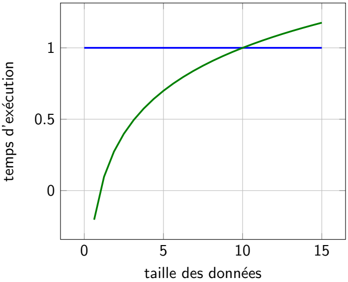
Complexité	Classe
$O(1)$	constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	sous-quadratique
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel



Quelques classes de complexité

- On peut donc "ranger" les fonctions équivalentes dans la même **classe** ;
- Voici quelques classes fréquentes de complexité (par ordre croissant en termes de $O(\cdot)$) :

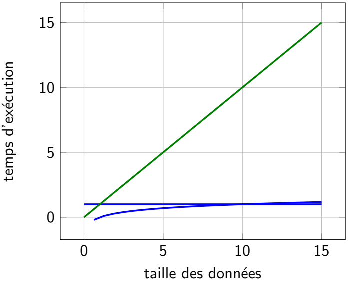
Complexité	**Classe**
$O(1)$	constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	sous-quadratique
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel



Quelques classes de complexité

- On peut donc "ranger" les fonctions équivalentes dans la même **classe** ;
- Voici quelques classes fréquentes de complexité (par ordre croissant en termes de $O(\cdot)$) :

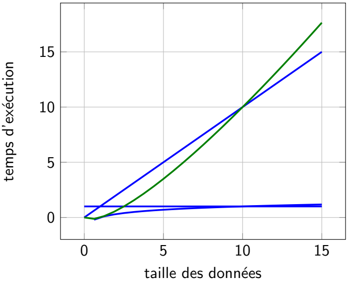
Complexité	Classe
$O(1)$	constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	sous-quadratique
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel



Quelques classes de complexité

- On peut donc "ranger" les fonctions équivalentes dans la même **classe** ;
- Voici quelques classes fréquentes de complexité (par ordre croissant en termes de $O(\cdot)$) :

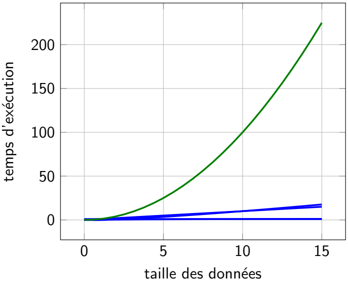
Complexité	**Classe**
$O(1)$	constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	sous-quadratique
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel



Quelques classes de complexité

- On peut donc "ranger" les fonctions équivalentes dans la même **classe** ;
- Voici quelques classes fréquentes de complexité (par ordre croissant en termes de $O(\cdot)$) :

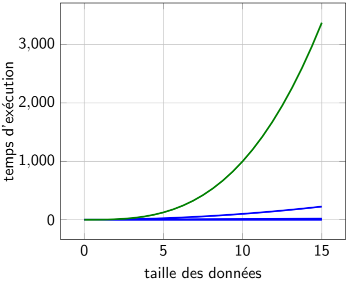
Complexité	**Classe**
$O(1)$	constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	sous-quadratique
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel



Quelques classes de complexité

- On peut donc "ranger" les fonctions équivalentes dans la même **classe** ;
- Voici quelques classes fréquentes de complexité (par ordre croissant en termes de $O(\cdot)$) :

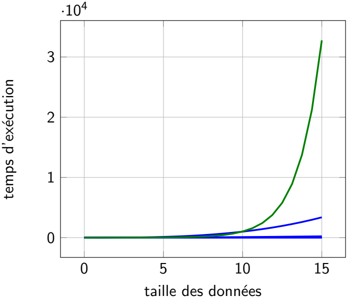
Complexité	**Classe**
$O(1)$	constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	sous-quadratique
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel



Quelques classes de complexité

- On peut donc "ranger" les fonctions équivalentes dans la même **classe** ;
- Voici quelques classes fréquentes de complexité (par ordre croissant en termes de $O(\cdot)$) :

Complexité	**Classe**
$O(1)$	constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	sous-quadratique
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel



Hiérarchie

- Pour faire un choix éclairé entre plusieurs algorithmes, il faut être capable de situer leur complexité ;

Hiérarchie

- Pour faire un choix éclairé entre plusieurs algorithmes, il faut être capable de situer leur complexité ;
- On fait une première distinction entre les deux classes suivantes :
  1. les algorithmes dits **polynomiaux**, dont la complexité est en $O(n^k)$ pour un certain k ;
  2. les algorithmes dits **exponentiels**, dont la complexité ne peut pas être majorée par une fonction polynomiale ;

Hiérarchie

- Pour faire un choix éclairé entre plusieurs algorithmes, il faut être capable de situer leur complexité ;
- On fait une première distinction entre les deux classes suivantes :
  1. les algorithmes dits **polynomiaux**, dont la complexité est en $O(n^k)$ pour un certain k ;
  2. les algorithmes dits **exponentiels**, dont la complexité ne peut pas être majorée par une fonction polynomiale ;
- De même :
  1. un problème de complexité polynomiale est considéré “facile” ;
  2. sinon (complexité non-polynomiale ou inconnue (!)) il est considéré “difficile” ;

Hiérarchie

- Pour faire un choix éclairé entre plusieurs algorithmes, il faut être capable de situer leur complexité ;
- On fait une première distinction entre les deux classes suivantes :
  1. les algorithmes dits **polynomiaux**, dont la complexité est en $O(n^k)$ pour un certain k ;
  2. les algorithmes dits **exponentiels**, dont la complexité ne peut pas être majorée par une fonction polynomiale ;
- De même :
  1. un problème de complexité polynomiale est considéré "facile" ;
  2. sinon (complexité non-polynomiale ou inconnue (!)) il est considéré "difficile" ;
- Mais quel est l'intérêt de la classification des problèmes ?

Intérêts de la classification de problèmes

Intérêts de la classification de problèmes

- (1) "Recyclage" de résultats ;

Intérêts de la classification de problèmes

- (1) "Recyclage" de résultats ;
- On peut parfois montrer que deux problèmes donnés :
  - sont équivalents ;
  - ou que l'on peut résoudre l'un à l'aide de l'autre ;

Intérêts de la classification de problèmes

- (1) "Recyclage" de résultats ;
- On peut parfois montrer que deux problèmes donnés :
  - sont équivalents ;
  - ou que l'on peut résoudre l'un à l'aide de l'autre ;
- Conséquences :
  - si on peut résoudre le problème P_1 en résolvant le problème P_2, alors on a directement un algorithme pour P_1 grâce à P_2 ;

Intérêts de la classification de problèmes

- (1) “Recyclage” de résultats ;
- On peut parfois montrer que deux problèmes donnés :
  - sont équivalents ;
  - ou que l'on peut résoudre l'un à l'aide de l'autre ;
- Conséquences :
  - si on peut résoudre le problème P_1 en résolvant le problème P_2, alors on a directement un algorithme pour P_1 grâce à P_2 ;
  - on a donc complexité$(P_1) \leq$ complexité(P_2) ;
  - et on ne peut donc pas résoudre P_2 plus vite que P_1 ;

Intérêts de la classification de problèmes

- (2) Savoir comment attaquer un problème selon sa catégorie ;

Intérêts de la classification de problèmes

- (2) Savoir comment attaquer un problème selon sa catégorie ;
- Certains problèmes sont particulièrement difficiles, et on ne peut pas espérer les résoudre en temps polynomial (cf. plus loin) ;

Intérêts de la classification de problèmes

- (2) Savoir comment attaquer un problème selon sa catégorie ;
- Certains problèmes sont particulièrement difficiles, et on ne peut pas espérer les résoudre en temps polynomial (cf. plus loin) ;
- Si on peut prouver qu'un problème appartient à cette catégorie, on a une meilleure idée des techniques qui ont une chance de marcher pour le résoudre en pratique ;

Intérêts de la classification de problèmes

- (3) Applications : cryptographie ;
- Exemple : le **cryptosystème de Rabin**, inventé en 1979 par Michael O. Rabin :

- S'appuie sur le problème de factorisation, qu'on suppose difficile ;

Fonctionnement du cryptosystème de Rabin

- Cryptosystème **asymétrique** : on chiffre avec une **clé publique** et on déchiffre à l'aide d'une **clé privée** ;

Fonctionnement du cryptosystème de Rabin

- Cryptosystème **asymétrique** : on chiffre avec une **clé publique** et on déchiffre à l'aide d'une **clé privée** ;
- Génération des clés :

Fonctionnement du cryptosystème de Rabin

- Cryptosystème **asymétrique** : on chiffre avec une **clé publique** et on déchiffre à l'aide d'une **clé privée** ;
- Génération des clés :
  1. choisir deux nombres premiers p et q, qui forment la clé privée ;
  2. la clé publique est le nombre $n = p * q$;

Fonctionnement du cryptosystème de Rabin

- Cryptosystème **asymétrique** : on chiffre avec une **clé publique** et on déchiffre à l'aide d'une **clé privée** ;
- Génération des clés :
  1. choisir deux nombres premiers p et q, qui forment la clé privée ;
  2. la clé publique est le nombre $n = p * q$;
- On chiffre les entrées en prenant leur carré modulo n ;

Fonctionnement du cryptosystème de Rabin

- Cryptosystème **asymétrique** : on chiffre avec une **clé publique** et on déchiffre à l'aide d'une **clé privée** ;
- Génération des clés :
  1. choisir deux nombres premiers p et q, qui forment la clé privée ;
  2. la clé publique est le nombre $n = p * q$;
- On chiffre les entrées en prenant leur carré modulo n ;
- On déchiffre le message en calculant les racines carrées modulo p et q ;

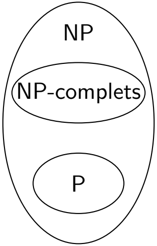
Fonctionnement du cryptosystème de Rabin

- Cryptosystème **asymétrique** : on chiffre avec une **clé publique** et on déchiffre à l'aide d'une **clé privée** ;
- Génération des clés :
  1. choisir deux nombres premiers p et q, qui forment la clé privée ;
  2. la clé publique est le nombre $n = p * q$;
- On chiffre les entrées en prenant leur carré modulo n ;
- On déchiffre le message en calculant les racines carrées modulo p et q ;

- Si on connaît la clé privée, le déchiffrement est facile ;
- Si on ne la connaît pas, il faut factoriser n ;
  - ... mais on ne connaît pas d'algorithme de complexité polynomiale en b (le nombre de bits de n) ;

Une question à un million de dollars

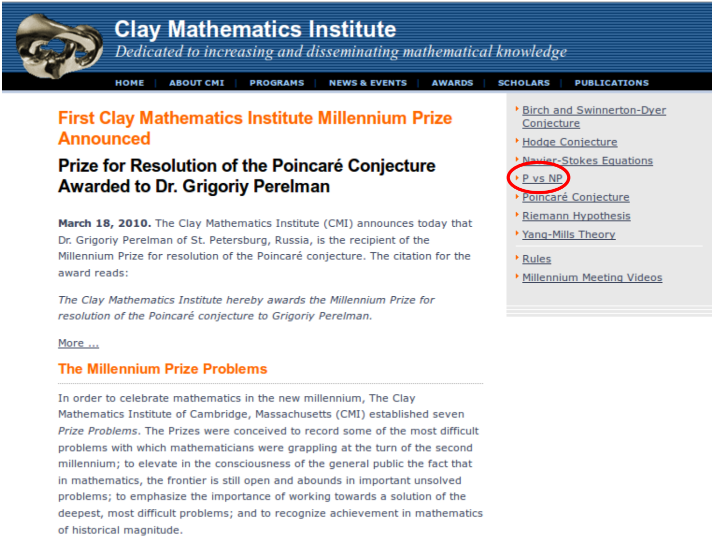
- **La classe** P est l'ensemble des problèmes qu'on peut résoudre avec un algorithme de complexité polynomiale ;
- **La classe** NP est l'ensemble des problèmes dont on peut **vérifier** une solution avec un algorithme de complexité polynomiale ;



- La question la plus importante en informatique théorique est :

$$P \,?\!= NP$$

La fondation Clay



Clay Mathematics Institute

Dedicated to increasing and disseminating mathematical knowledge

HOME | ABOUT CMI | PROGRAMS | NEWS & EVENTS | AWARDS | SCHOLARS | PUBLICATIONS

First Clay Mathematics Institute Millennium Prize Announced

Prize for Resolution of the Poincaré Conjecture Awarded to Dr. Grigoriy Perelman

March 18, 2010. The Clay Mathematics Institute (CMI) announces today that Dr. Grigoriy Perelman of St. Petersburg, Russia, is the recipient of the Millennium Prize for resolution of the Poincaré conjecture. The citation for the award reads:

The Clay Mathematics Institute hereby awards the Millennium Prize for resolution of the Poincaré conjecture to Grigoriy Perelman.

More ...

The Millennium Prize Problems

In order to celebrate mathematics in the new millennium, The Clay Mathematics Institute of Cambridge, Massachusetts (CMI) established seven *Prize Problems*. The Prizes were conceived to record some of the most difficult problems with which mathematicians were grappling at the turn of the second millennium; to elevate in the consciousness of the general public the fact that in mathematics, the frontier is still open and abounds in important unsolved problems; to emphasize the importance of working towards a solution of the deepest, most difficult problems; and to recognize achievement in mathematics of historical magnitude.

- Birch and Swinnerton-Dyer Conjecture
- Hodge Conjecture
- Navier-Stokes Equations
- P vs NP
- Poincaré Conjecture
- Riemann Hypothesis
- Yang-Mills Theory

- Rules
- Millennium Meeting Videos

http://www.claymath.org/millennium/

Le(s) prix

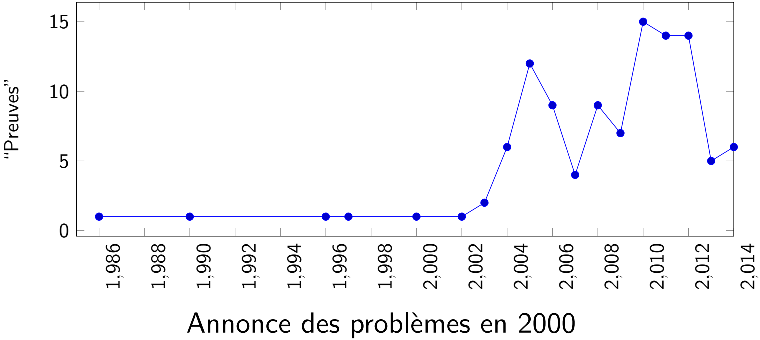
- Extrait de `http://www.claymath.org/millennium/Rules_etc/` :

"(...) The Clay Mathematics Institute (CMI) of Cambridge, Massachusetts has named seven "Millennium Prize Problems." (...) **The Board of Directors of CMI designated a $7 million prize fund for the solution to these problems, with $1 million allocated to each.** *(...)"*

- Si vous prouvez P=NP ou P≠NP, c'est gagné ;
- Si c'est trop difficile, vous pouvez toujours essayer les autres problèmes ;

"Preuves" de P=NP et de P$\neq$ NP

- Les chercheurs et les amateurs n'ont pas attendu ce prix pour tenter de résoudre cette question ;



(Source : "The P-versus-NP page", Gerhard J. Woeginger, 02/09/2014 – `http://www.win.tue.nl/~gwoegi/P-versus-NP.htm`)

Quelques problèmes faciles

- Rechercher un/le plus petit/le plus grand élément dans une base de données ;
- Trier une base de données ;
- Rechercher les occurrences d'un motif dans un texte ;
- Calculer l'itinéraire le plus court entre deux sommets d'un graphe ;

Quelques problèmes difficiles (NP-complets)

Problème (SUBSET SUM)

Données : un ensemble S de n entiers ;
Question : existe-t-il un sous-ensemble $S' \subseteq S$ dont la somme des éléments est nulle ?

- Exemple : $S = \{-7, -3, -2, 5, 8\}$; $S' = \{-3, -2, 5\}$ est une solution ;

Quelques problèmes difficiles (NP-complets)

Problème (SUBSET SUM)

Données : un ensemble S de n entiers ;
Question : existe-t-il un sous-ensemble $S' \subseteq S$ dont la somme des éléments est nulle ?

- Exemple : $S = \{-7, -3, -2, 5, 8\}$; $S' = \{-3, -2, 5\}$ est une solution ;

Problème (PARTITION)

Données : un ensemble S de nombres (répétitions autorisées) ;
Question : peut-on séparer S en deux sous-ensembles A et B tels que $\sum_{a \in A} a = \sum_{b \in B} b$?

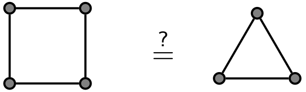
- Si $S = \{1, 3, 1, 2, 2, 1\}$, alors $A = \{1, 3, 1\}$ et $B = \{1, 2, 2\}$ marche ;

Un problème de complexité inconnue

Problème (ISOMORPHISME DE GRAPHES)

Données : deux graphes G_1 et G_2 ;
Question : G_1 et G_2 sont-ils "les mêmes" ? (peut-on numéroter leurs sommets de manière à obtenir deux ensembles d'arêtes identiques ?)

Exemple

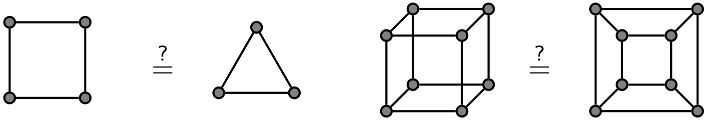


Un problème de complexité inconnue

Problème (ISOMORPHISME DE GRAPHES)

Données : deux graphes G_1 et G_2 ;
Question : G_1 et G_2 sont-ils “les mêmes” ? (peut-on numéroter leurs sommets de manière à obtenir deux ensembles d'arêtes identiques ?)

Exemple



Un problème de complexité inconnue

Problème (ISOMORPHISME DE GRAPHES)

Données : deux graphes G_1 et G_2 ;
Question : G_1 et G_2 sont-ils "les mêmes" ? (peut-on numéroter leurs sommets de manière à obtenir deux ensembles d'arêtes identiques ?)

Exemple