

Chapitre 1

Introduction

HLIN401 : Algorithmique et complexité

L2 Informatique
Université de Montpellier
2020 – 2021

Déroulement du cours

- Responsable : B. Grenet (bruno.grenet@umontpellier.fr)

Déroulement du cours

- ▶ Responsable : B. Grenet (bruno.grenet@umontpellier.fr)
- ▶ Cours : lundi 9h45-11h15, BBB ou amphi 6.04 (*sauf changement*)
- ▶ TD/TP : par groupes (*cf* emploi du temps en ligne)
 - A+CMI B. Grenet
 - B E. Gurpinar
 - C B. Grenet
 - D A. Perret du Cray

Déroulement du cours

- ▶ Responsable : B. Grenet (bruno.grenet@umontpellier.fr)
- ▶ Cours : lundi 9h45-11h15, BBB ou amphi 6.04 (*sauf changement*)
- ▶ TD/TP : par groupes (*cf* emploi du temps en ligne)
 - A+CMI B. Grenet
 - B E. Gurpinar
 - C B. Grenet
 - D A. Perret du Cray
- ▶ Évaluations : CC + Examen. $\text{Note} = \max(E, 30\%CC + 70\%E)$
 - ▶ Partiel type examen (15 pts) : *a priori* le 8 mars
 - ▶ TP (5 pts) : **tous** les TPs à rendre, via Moodle, avec correction automatique et détection de plagiat

Déroulement du cours

- ▶ Responsable : B. Grenet (bruno.grenet@umontpellier.fr)
- ▶ Cours : lundi 9h45-11h15, BBB ou amphi 6.04 (*sauf changement*)
- ▶ TD/TP : par groupes (*cf* emploi du temps en ligne)
 - A+CMI B. Grenet
 - B E. Gurpinar
 - C B. Grenet
 - D A. Perret du Cray
- ▶ Évaluations : CC + Examen. $\text{Note} = \max(E, 30\%CC + 70\%E)$
 - ▶ Partiel type examen (15 pts) : *a priori* le 8 mars
 - ▶ TP (5 pts) : **tous** les TPs à rendre, via Moodle, avec correction automatique et détection de plagiat
- ▶ **Moodle – cours HLIN401 : à consulter impérativement !**

Moodle encore plus important !

Moodle encore plus important !

- ▶ Cours / TD : visio-conférences BBB
- ▶ Séances enregistrées

Moodle encore plus important !

- ▶ Cours / TD : visio-conférences BBB
- ▶ Séances enregistrées
- ▶ TP :
 - ▶ Sujets connus à l'avance
 - ▶ Code à rendre (avec détection de plagiat) et noté
 - ▶ Travail en autonomie + séances d'aide (visio, chat, etc.)

Il est très fortement recommandé de travailler le TP avant la séance d'aide

1. Exemple introductif : calculer x^n

2. Modèle pour la complexité algorithmique

3. Conception et analyse d'un algorithme

4. Outils mathématiques

Le problème

Pour un réel x et un entier $n \geq 1$, on veut calculer x^n

Le problème

Pour un réel x et un entier $n \geq 1$, on veut calculer x^n

Pour cela on va

- ▶ proposer plusieurs **algorithmes** et les analyser :
- ▶ démontrer leur **correction**
- ▶ estimer leur **complexité**
(= temps nécessaire au déroulement du programme)
- ▶ voir une implémentation possible

Le problème

Pour un réel x et un entier $n \geq 1$, on veut calculer x^n

Pour cela on va

- ▶ proposer plusieurs **algorithmes** et les analyser :
- ▶ démontrer leur **correction**
- ▶ estimer leur **complexité**
(= temps nécessaire au déroulement du programme)
- ▶ voir une implémentation possible

Remarque. Problème très utile en pratique !

(résolution d'équa. diff., codes correcteurs, crypto : RSA, courbes elliptiques...)

Algo 1 : multiplications successives

Algorithme : $\text{ALGOIT}(x, n)$

y un réel

$y \leftarrow x$

pour tous les i *de 1 à $n - 1$* **faire**

$y \leftarrow x \times y$

renvoyer y

Algo 1 : multiplications successives

Algorithme : $\text{ALGOLT}(x, n)$

y un réel

$y \leftarrow x$

pour tous les i *de 1 à $n - 1$* **faire**

$y \leftarrow x \times y$

renvoyer y

Un petit exemple :

On effectue **l'appel** $\text{ALGOLT}(2, 5)$:

- Initialisation de y : $y = 2$
- Étape $i = 1$: $y = 2 \times 2 = 4$
- Étape $i = 2$: $y = 2 \times 4 = 8$
- Étape $i = 3$: $y = 2 \times 8 = 16$
- Étape $i = 4$: $y = 2 \times 16 = 32$
- **L'algo retourne 32**

Algo 1 : multiplications successives

Algorithme : $\text{ALGOLT}(x, n)$

y un réel

$y \leftarrow x$

pour tous les i *de 1 à $n - 1$* **faire**

$y \leftarrow x \times y$

renvoyer y

Terminaison

À la fin de la boucle **pour**, l'algo. termine.

Algo 1 : multiplications successives

Algorithme : $\text{ALGOIT}(x, n)$

y un réel

$y \leftarrow x$

pour tous les i *de* 1 à $n - 1$ **faire**

$y \leftarrow x \times y$

renvoyer y

Complexité (en temps)

Nombre d'opérations élémentaires :

- Récupérations de x et n , déclaration de y ;
affectation ($y \leftarrow x$), retour \rightsquigarrow **5 op.**
- Dans la boucle **pour** : incrémentation de i ;
multiplication et affectation ($y \leftarrow x \times y$) \rightsquigarrow **3 op.**
- $n - 1$ répétitions de la boucle \rightsquigarrow **$3n + 2$ op.**

\rightsquigarrow **Complexité en temps $O(n)$**

Algo 1 : multiplications successives

Algorithme : $\text{ALGOIT}(x, n)$

y un réel

$y \leftarrow x$

pour tous les i *de* 1 *à* $n - 1$ **faire**

$y \leftarrow x \times y$

renvoyer y

Correction : preuve d'un **invariant de l'algo.**

\mathcal{P}_i : « après i tours de boucle, y contient x^{i+1} »

Algo 1 : multiplications successives

Algorithme : $\text{ALGOIT}(x, n)$

y un réel

$y \leftarrow x$

pour tous les i *de* 1 à $n - 1$ **faire**

$y \leftarrow x \times y$

renvoyer y

Correction : preuve d'un **invariant de l'algo**.

\mathcal{P}_i : « après i tours de boucle, y contient x^{i+1} »

Preuve par **récurrence** (quelle surprise...) :

- Pour $i = 0$, \mathcal{P}_0 est vraie : avant la boucle, y vaut x ($= x^1$).
- Supposons \mathcal{P}_{i-1} pour $i \geq 1$: après $(i - 1)$ tours, y contient $x^{(i-1)+1} = x^i$. Alors au $i^{\text{ème}}$ tour, y prend la valeur $x \times y = x^{i+1}$.
Donc \mathcal{P}_i est vraie.

Donc, par récurrence, $y = x^n$ à la fin de l'algo. ■

Algo 1 : multiplications successives

Algorithme : $\text{ALGOIT}(x, n)$

y un réel

$y \leftarrow x$

pour tous les i *de 1 à $n - 1$* **faire**

$y \leftarrow x \times y$

renvoyer y

Exemple de code :

```
float AlgoIt(float x, int n)
{
    float y;
    y = x;
    for (int i = 1; i < n; i++)
        y *= x;
    return y;
}

int main(int argc, char** argv)
{
    float x = strtod(argv[1], NULL);
    int n = strtol(argv[2], NULL, 0);
    cout << AlgoIt(x, n) << endl;
    return 0;
}
```

Algo 2 : Diviser pour régner

Algorithme : ALGO D&C(x, n)

si $n = 1$: **renvoyer** x

sinon :

$z = \text{ALGO D\&C}(x, \lfloor n/2 \rfloor)$

si n est pair : **renvoyer** $z \times z$

si n est impair : **renvoyer** $x \times z \times z$

Algo 2 : Diviser pour régner

Algorithme : $\text{ALGO D\&C}(x, n)$

si $n = 1$: **renvoyer** x

sinon :

$z = \text{ALGO D\&C}(x, \lfloor n/2 \rfloor)$

si n est pair : **renvoyer** $z \times z$

si n est impair : **renvoyer** $x \times z \times z$

Un petit exemple :

On effectue l'appel $\text{ALGO D\&C}(3, 5)$:

- $\text{ALGO D\&C}(3, 5)$ calcule $z = \text{ALGO D\&C}(3, 2)$ et retourne $3z^2$
- $\text{ALGO D\&C}(3, 2)$ calcule $z = \text{ALGO D\&C}(3, 1)$ et retourne z^2
- $\text{ALGO D\&C}(3, 1)$ retourne 3

Algo 2 : Diviser pour régner

Algorithme : $\text{ALGO D\&C}(x, n)$

si $n = 1$: renvoyer x

sinon :

$z = \text{ALGO D\&C}(x, \lfloor n/2 \rfloor)$

si n est pair : renvoyer $z \times z$

si n est impair : renvoyer $x \times z \times z$

Un petit exemple :

On effectue l'appel $\text{ALGO D\&C}(3, 5)$:

- $\text{ALGO D\&C}(3, 5)$ calcule $z = \text{ALGO D\&C}(3, 2)$ et retourne $3z^2$
- $\text{ALGO D\&C}(3, 2)$ calcule $z = \text{ALGO D\&C}(3, 1)$ et retourne z^2
- $\text{ALGO D\&C}(3, 1)$ retourne 3
- Du coup, $\text{ALGO D\&C}(3, 2)$ retourne $3^2 = 9$
- Du coup, $\text{ALGO D\&C}(3, 5)$ retourne $3 \times 9^2 = 243$

Algo 2 : Diviser pour régner

Algorithme : ALGO D&C(x, n)

si $n = 1$: **renvoyer** x

sinon :

$z = \text{ALGO D\&C}(x, \lfloor n/2 \rfloor)$

si n est pair : **renvoyer** $z \times z$

si n est impair : **renvoyer** $x \times z \times z$

Terminaison

- ▶ **Nombre constant d'opérations** (≤ 5) + un appel récursif
- ▶ Appel récursif sur un **paramètre strictement plus petit** mais toujours ≥ 1 (*variant* de l'algo.)
- ▶ **Cas de base** présent

↪ L'algorithme termine.

Algo 2 : Diviser pour régner

Algorithme : ALGO D&C(x, n)

si $n = 1$: **renvoyer** x

sinon :

$z = \text{ALGO D\&C}(x, \lfloor n/2 \rfloor)$

si n est pair : **renvoyer** $z \times z$

si n est impair : **renvoyer** $x \times z \times z$

Complexité

Nombre constant d'opérations

↪ complexité **proportionnelle** au nombre d'appels récurifs

Algo 2 : Diviser pour régner

Algorithme : $\text{ALGO D\&C}(x, n)$

si $n = 1$: **renvoyer** x

sinon :

$z = \text{ALGO D\&C}(x, \lfloor n/2 \rfloor)$

si n est pair : **renvoyer** $z \times z$

si n est impair : **renvoyer** $x \times z \times z$

Nombre d'appels récursifs (nb de fois qu'on peut diviser n par 2 $\rightsquigarrow \log n$)

\mathcal{P}_n : « $\text{ALGO D\&C}(x, n)$ fait au plus $\log n$ appels récursifs »

Algo 2 : Diviser pour régner

Algorithme : ALGOD&C(x, n)

si $n = 1$: **renvoyer** x

sinon :

$z = \text{ALGOD\&C}(x, \lfloor n/2 \rfloor)$

si n est pair : **renvoyer** $z \times z$

si n est impair : **renvoyer** $x \times z \times z$

Nombre d'appels récurrents (nb de fois qu'on peut diviser n par 2 $\rightsquigarrow \log n$)

\mathcal{P}_n : « ALGOD&C(x, n) fait au plus $\log n$ appels récurrents »

- ▶ $n = 1$: aucun appel récursif et $\log(1) = 0$
- ▶ Soit $n \geq 2$ et supposons \mathcal{P}_p **pour tout** $p < n$: le nombre d'appels de ALGOD&C($x, \lfloor \frac{n}{2} \rfloor$) est au plus $\log(\lfloor \frac{n}{2} \rfloor) \leq \log(\frac{n}{2}) = \log(n) - 1$. Donc le nombre d'appels de ALGOD&C(x, n) est $\leq 1 + (\log(n) - 1) = \log(n)$.

\rightsquigarrow **Complexité au plus proportionnelle à $\log n$ (en $O(\log n)$)**

Algo 2 : Diviser pour régner

Algorithme : ALGO D&C(x, n)

si $n = 1$: **renvoyer** x

sinon :

$z = \text{ALGO D\&C}(x, \lfloor n/2 \rfloor)$

si n est pair : **renvoyer** $z \times z$

si n est impair : **renvoyer** $x \times z \times z$

Correction :

$\mathcal{P}_n : \ll \text{ALGO D\&C}(x, n) \text{ renvoie } x^n \gg$

Algo 2 : Diviser pour régner

Algorithme : ALGOD&C(x, n)

si $n = 1$: **renvoyer** x

sinon :

$z = \text{ALGOD\&C}(x, \lfloor n/2 \rfloor)$

si n est pair : **renvoyer** $z \times z$

si n est impair : **renvoyer** $x \times z \times z$

Correction : $\mathcal{P}_n : \ll \text{ALGOD\&C}(x, n) \text{ renvoie } x^n \gg$

► $n = 1$: $\text{ALGOD\&C}(x, 1)$ renvoie $x \rightsquigarrow \mathcal{P}_1$ est vraie

► Soit $n \geq 2$ et supposons \mathcal{P}_p pour tout $p < n$: $\text{ALGOD\&C}(x, \lfloor n/2 \rfloor)$ renvoie $z = x^{\lfloor n/2 \rfloor}$ (car $\lfloor n/2 \rfloor < n$!).

► Si n est pair : $n = 2 \times \lfloor n/2 \rfloor$ et $\text{ALGOD\&C}(x, n)$ renvoie $z \times z = x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} = x^n$.

► Si n est impair, $n = 2 \times \lfloor n/2 \rfloor + 1$ et $\text{ALGOD\&C}(x, n)$ renvoie $x \times z \times z = x^{2\lfloor n/2 \rfloor + 1} = x^n$.

Donc \mathcal{P}_n est vraie.

Algo 2 : Diviser pour régner

Algorithme : ALGO D&C(x, n)

si $n = 1$: renvoyer x

sinon :

$z = \text{ALGO D\&C}(x, \lfloor n/2 \rfloor)$

si n est pair : renvoyer $z \times z$

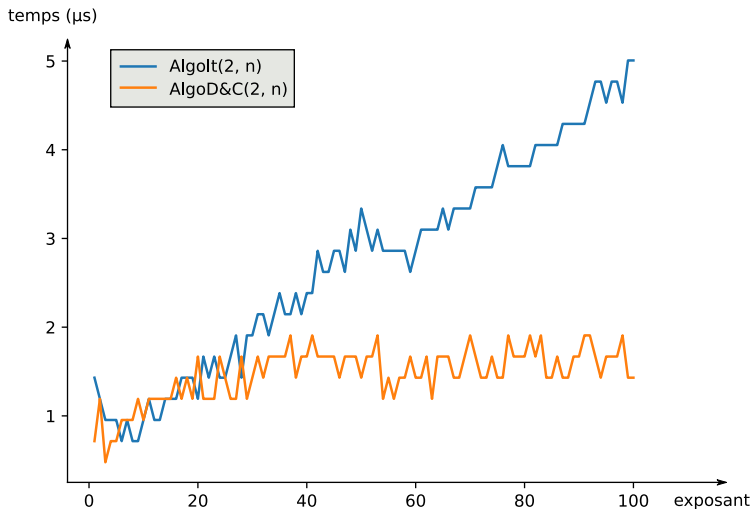
si n est impair : renvoyer $x \times z \times z$

Exemple de code :

```
float AlgoDC(float x, int n)
{
    if (n == 1) return x;
    float z = AlgoDC(x, n/2);
    if (n % 2 == 0) return z*z;
    /* else */ return x*z*z;
}
```

```
int main(int argc, char** argv)
{
    float x = strtod(argv[1], NULL);
    int n = strtol(argv[2], NULL, 0);
    cout << AlgoDC(x, n) << endl;
    return 0;
}
```

Comparaison : ALGOLT vs. ALGOD&C



Remarque : $O(n)$ croît plus vite que $O(\log n)$...

Algo 3 : Arnaque

Algorithme : ALGOARNAQUE(x, n)
renvoyer $\text{pow}(x, n)$

Exemple de code :

```
#include <math.h>

float AlgoArnaque(float x, int n)
{
    return pow(x, n);
}

int main(int argc, char** argv)
{
    float x = strtod(argv[1], NULL);
    int n = strtol(argv[2], NULL, 0);
    cout << AlgoArnaque(x, n) << endl;
    return 0;
}
```

Algo 3 : Arnaque

Algorithme : `ALGOARNAQUE(x, n)`
renvoyer `pow(x, n)`

- ▶ Très pratique... mais qu'y a-t-il dessous ?
- ▶ Quelques idées :
 - ▶ <https://en.cppreference.com/w/cpp/numeric/math/pow>
 - ▶ <https://www.quora.com/What-is-the-time-complexity-of-the-pow-function-in-c++-language-Is-it-log-b-or-0-1>

std::pow, std::powf, std::powl

Defined in header `<cmath>`

float	pow (float base, float exp);	(1)	
float	powf(float base, float exp);		(since C++11)
double	pow (double base, double exp);	(2)	
long double	pow (long double base, long double exp);	(3)	
long double	powl(long double base, long double exp);		(since C++11)
float	pow (float base, int iexp);	(4)	(until C++11)
double	pow (double base, int iexp);	(5)	(until C++11)
long double	pow (long double base, int iexp);	(6)	(until C++11)
Promoted	pow (Arithmetic1 base, Arithmetic2 exp);	(7)	(since C++11)

1-6) Computes the value of base raised to the power exp or iexp.

7) A set of overloads or a function template for all combinations of arguments of arithmetic type not covered by 1-3). If any argument has [integral type](#), it is cast to `double`. If any argument is `long double`, then the return type Promoted is also `long double`, otherwise the return type is always `double`.

Parameters

Algo 3 : Arnaque

Algorithme : `ALGOARNAQUE(x, n)`
renvoyer `pow(x, n)`

- ▶ Très pratique... mais qu'y a-t-il dessous ?
- ▶ Quelques idées :
 - ▶ <https://en.cppreference.com/w/cpp/numeric/math/pow>
 - ▶ <https://www.quora.com/What-is-the-time-complexity-of-the-pow-function-in-c++-language-Is-it-log-b-or-0-1>
- ▶ Si on veut vraiment savoir, il faut analyser le code de `pow`...

1. Exemple introductif : calculer x^n
2. Modèle pour la complexité algorithmique
3. Conception et analyse d'un algorithme
4. Outils mathématiques

Pourquoi un modèle ?

Objectif : répondre à la question « *Quel temps va nécessiter la résolution d'un problème algorithmique ?* »

Pourquoi un modèle ?

Objectif : répondre à la question « *Quel temps va nécessiter la résolution d'un problème algorithmique ?* »

- ▶ Difficile à estimer : dépend du programme, du langage, de la machine, du système d'exploitation...

Pourquoi un modèle ?

Objectif : répondre à la question « *Quel temps va nécessiter la résolution d'un problème algorithmique ?* »

- ▶ Difficile à estimer : dépend du programme, du langage, de la machine, du système d'exploitation...
- ▶ Considérer un modèle permet de faire des *prédictions*

Pourquoi un modèle ?

Objectif : répondre à la question « *Quel temps va nécessiter la résolution d'un problème algorithmique ?* »

- ▶ Difficile à estimer : dépend du programme, du langage, de la machine, du système d'exploitation...
- ▶ Considérer un modèle permet de faire des *prédictions*

L'étude de la complexité est une **modélisation** permettant des prédictions.

Modèle choisi

On va décrire les algorithmes en **pseudo-code** :

► Des **opérations élémentaires** :

- Déclaration de variable
- Affectation
- Lecture, écriture de variables
- Opération arithmétique : $+$, $-$, \times , \div
- Test élémentaire
- Appel de fonction

► Des **branchements** : *si ... alors ... sinon ...*

► Des **boucles** : *pour* et *tant que*.

Modèle choisi

On va décrire les algorithmes en **pseudo-code** :

► Des **opérations élémentaires** :

- Déclaration de variable
- Affectation
- Lecture, écriture de variables
- Opération arithmétique : $+$, $-$, \times , \div
- Test élémentaire
- Appel de fonction

► Des **branchements** : *si ... alors ... sinon ...*

► Des **boucles** : *pour* et *tant que*.

► Deux modèles :

Word-RAM : chaque **opération élémentaire** prend un **temps constant**

Modèle choisi

On va décrire les algorithmes en **pseudo-code** :

► Des **opérations élémentaires** :

- Déclaration de variable
- Affectation
- Lecture, écriture de variables
- **Opération arithmétique** : $+$, $-$, \times , \div
- Test élémentaire
- Appel de fonction

► Des **branchements** : *si ... alors ... sinon ...*

► Des **boucles** : *pour* et *tant que*.

► Deux modèles :

Word-RAM : chaque **opération élémentaire** prend un **temps constant**

RAM : chaque **opération sur un bit/chiffre** prend un **temps constant**

Modèle choisi

On va décrire les algorithmes en **pseudo-code** :

► Des **opérations élémentaires** :

- Déclaration de variable
- Affectation
- Lecture, écriture de variables
- **Opération arithmétique** : $+$, $-$, \times , \div
- Test élémentaire
- Appel de fonction

► Des **branchements** : *si ... alors ... sinon ...*

► Des **boucles** : *pour* et *tant que*.

► Deux modèles :

Word-RAM : chaque **opération élémentaire** prend un **temps constant**

RAM : chaque **opération sur un bit/chiffre** prend un **temps constant**

► Temps pour lire un entier n , dans chaque modèle ?

Définition de la complexité

Sauf cas particulier, modèle Word-RAM.

- **Compter le nombre d'opérations élémentaires** (pour établir la **complexité en temps**)

Définition de la complexité

Sauf cas particulier, modèle Word-RAM.

- ▶ **Compter le nombre d'opérations élémentaires** (pour établir la **complexité en temps**)
- ▶ Exprimer ces valeurs **en fonction des paramètres d'entrée** de l'algorithme.

Définition de la complexité

Sauf cas particulier, modèle Word-RAM.

- ▶ **Compter le nombre d'opérations élémentaires** (pour établir la **complexité en temps**)
- ▶ Exprimer ces valeurs **en fonction des paramètres d'entrée** de l'algorithme.
- ▶ De manière **asymptotique**

Définition de la complexité

Sauf cas particulier, modèle Word-RAM.

- ▶ **Compter le nombre d'opérations élémentaires** (pour établir la **complexité en temps**)
- ▶ Exprimer ces valeurs **en fonction des paramètres d'entrée** de l'algorithme.
- ▶ De manière **asymptotique**
- ▶ Dans le **pire des cas**, et si on n'arrive pas à compter exactement, on établira une borne supérieure sur ces valeurs.

Quelques remarques

Étudier la complexité **en temps** dans le pire cas de manière asymptotique dans le modèle **Word-RAM** : la seule façon de faire ?

Quelques remarques

Étudier la complexité **en temps dans le pire cas de manière asymptotique dans le modèle Word-RAM** : la seule façon de faire ? **Non** :

- ▶ **Temps** : autres mesures (espace mémoire, temps parallèle, ...)
- ▶ **Pire cas** : raffinements possibles (en moyenne, analyses amortie et lissée, cas pratiques, ...)
- ▶ **Asymptotique** : constantes « cachées » dans les $O(\cdot)$ mais en pratique, elles peuvent avoir leur importance...
- ▶ **Modèle Word-RAM** : autres modèles, par ex. RAM

Quelques remarques

Étudier la complexité **en temps dans le pire cas de manière asymptotique dans le modèle Word-RAM** : la seule façon de faire ? **Non** :

- ▶ **Temps** : autres mesures (espace mémoire, temps parallèle, ...)
- ▶ **Pire cas** : raffinements possibles (en moyenne, analyses amortie et lissée, cas pratiques, ...)
- ▶ **Asymptotique** : constantes « cachées » dans les $O(\cdot)$ mais en pratique, elles peuvent avoir leur importance...
- ▶ **Modèle Word-RAM** : autres modèles, par ex. RAM

Modèle choisi dans le cours car **le plus simple** et :

- ▶ souvent suffisant
- ▶ on commence toujours par ça
- ▶ si on comprend ce modèle, on comprendra (plus tard !) les autres

1. Exemple introductif : calculer x^n
2. Modèle pour la complexité algorithmique
3. Conception et analyse d'un algorithme
4. Outils mathématiques

Conception et analyse d'un algorithme

« Recette » :

1. Écrire le **pseudo-code** de l'algorithme
2. Choisir les **structures de données** à utiliser pour les variables (influence la complexité de l'algo !)
On va peu s'en préoccuper dans ce cours.
3. **Analyser l'algorithme** :

Conception et analyse d'un algorithme

« Recette » :

1. Écrire le **pseudo-code** de l'algorithme
2. Choisir les **structures de données** à utiliser pour les variables (influence la complexité de l'algo !)

On va peu s'en préoccuper dans ce cours.

3. Analyser l'algorithme :

3.1 Terminaison

3.2 Complexité en temps

3.3 Correction de l'algorithme

Conception et analyse d'un algorithme

« Recette » :

1. Écrire le **pseudo-code** de l'algorithme
2. Choisir les **structures de données** à utiliser pour les variables (influence la complexité de l'algo !)

On va peu s'en préoccuper dans ce cours.

3. Analyser l'algorithme :

3.1 Terminaison

- ▶ **Variant** d'algorithme = quantité entière positive qui décroît strictement au cours de l'algo.

3.2 Complexité en temps

3.3 Correction de l'algorithme

Conception et analyse d'un algorithme

« Recette » :

1. Écrire le **pseudo-code** de l'algorithme
2. Choisir les **structures de données** à utiliser pour les variables (influence la complexité de l'algo !)

On va peu s'en préoccuper dans ce cours.

3. Analyser l'algorithme :

3.1 Terminaison

- ▶ **Variant** d'algorithme = quantité entière positive qui décroît strictement au cours de l'algo.
- ▶ Souvent omise \rightsquigarrow clair avec complexité et correction

3.2 Complexité en temps

3.3 Correction de l'algorithme

Conception et analyse d'un algorithme

« Recette » :

1. Écrire le **pseudo-code** de l'algorithme
2. Choisir les **structures de données** à utiliser pour les variables (influence la complexité de l'algo !)

On va peu s'en préoccuper dans ce cours.

3. Analyser l'algorithme :

3.1 Terminaison

- ▶ **Variant** d'algorithme = quantité entière positive qui décroît strictement au cours de l'algo.
- ▶ Souvent omise \rightsquigarrow clair avec complexité et correction

3.2 Complexité en temps

- ▶ **Borne supérieure** dans le **pire cas** : « Je suis sûr que mon algo ne prendra pas plus de ... »

3.3 Correction de l'algorithme

Conception et analyse d'un algorithme

« Recette » :

1. Écrire le **pseudo-code** de l'algorithme
2. Choisir les **structures de données** à utiliser pour les variables (influence la complexité de l'algo !)

On va peu s'en préoccuper dans ce cours.

3. Analyser l'algorithme :

3.1 Terminaison

- ▶ **Variant** d'algorithme = quantité entière positive qui décroît strictement au cours de l'algo.
- ▶ Souvent omise \rightsquigarrow clair avec complexité et correction

3.2 Complexité en temps

- ▶ **Borne supérieure** dans le **pire cas** : « Je suis sûr que mon algo ne prendra pas plus de ... »

3.3 Correction de l'algorithme

- ▶ **Invariant** d'algorithme = propriété \mathcal{P}_i valable après i tours de boucles / i appels récurifs.
- ▶ Preuve par **récurrence**

Correction d'un algorithme

- ▶ C'est souvent le plus technique à faire
- ▶ Mais c'est nécessaire !

Correction d'un algorithme

- ▶ C'est souvent le plus technique à faire
- ▶ Mais c'est nécessaire !

Exemple (popularisé par G. Berry) : dans l'appli *Zune* (lancée en 2006 sur les lecteurs MP3 Microsoft), une fonction donnait le numéro du jour de l'année à partir du nombre de jours écoulés depuis le 1er janvier 2004. Les lecteurs se sont déchargés entièrement le 31 décembre 2008 (jour 1827). Pourquoi ?

Correction d'un algorithme

Version simplifiée :

```
int JourDeLAnnee(int jour, int annee = 2004) {  
    while (jour > 365) {  
        if (estBissextile(annee)) {  
            if (jour > 366) {  
                jour -= 366;  
                annee += 1;  
            }  
        }  
        else {  
            jour -= 365;  
            annee += 1;  
        }  
    }  
    return jour;  
}
```

$$1827 = 366 + 365 + 365 + 365 + 366$$

Correction d'un algorithme

- ▶ C'est souvent le plus technique à faire
- ▶ Mais c'est nécessaire !

Exemple (popularisé par G. Berry) : dans l'appli *Zune* (lancée en 2006 sur les lecteurs MP3 Microsoft), une fonction donnait le numéro du jour de l'année à partir du nombre de jours écoulés depuis le 1er janvier 2004. Les lecteurs se sont déchargés entièrement le 31 décembre 2008 (jour 1827). Pourquoi ?

- ▶ D'autres exemples : https://en.wikipedia.org/wiki/List_of_software_bugs

1. Exemple introductif : calculer x^n
2. Modèle pour la complexité algorithmique
3. Conception et analyse d'un algorithme
4. Outils mathématiques

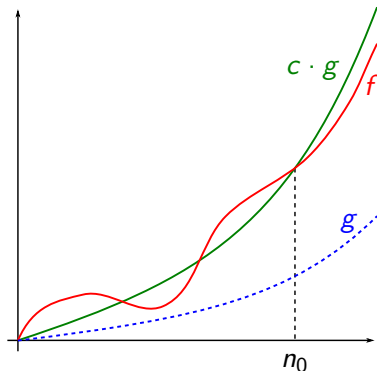
Notations de Landau

« Grand O »

Soit $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$. Alors $f = O(g)$ si

$$\exists c > 0, n_0 \geq 0, \forall n \geq n_0, f(n) \leq c \cdot g(n).$$

« f est un grand O de g s'il existe une constante c et un entier n_0 tels que pour toute valeur n plus grande que n_0 , $f(n)$ est inférieur ou égal à $c \cdot g(n)$ »



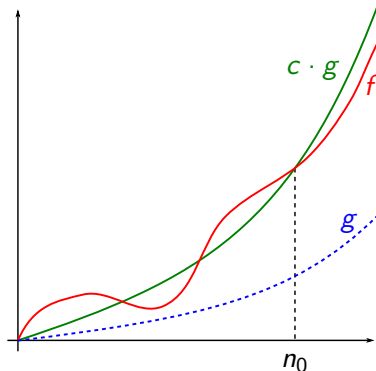
Notations de Landau

« Grand O »

Soit $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$. Alors $f = O(g)$ si

$$\exists c > 0, n_0 \geq 0, \forall n \geq n_0, f(n) \leq c \cdot g(n).$$

« f est un grand O de g s'il existe une constante c et un entier n_0 tels que pour toute valeur n plus grande que n_0 , $f(n)$ est inférieur ou égal à $c \cdot g(n)$ »



$f = O(g)$ si pour n suffisamment grand, f est plus petite que g , à une constante multiplicative près.

Utilisation en complexité

« Mon algo. a une complexité $O(n^2)$ (où n = taille de l'entrée) »

↪ si n est assez grand, le nb. d'opérations est $\leq \text{constante} \times n^2$

Utilisation en complexité

« Mon algo. a une complexité $O(n^2)$ (où n = taille de l'entrée) »

↪ si n est assez grand, le nb. d'opérations est $\leq \text{constante} \times n^2$

► Avantages pour la théorie :

- Négliger les cas de bases
- Pas besoin de compter chaque opération en détail
- Flexibilité sur les opérations élémentaires

Utilisation en complexité

« Mon algo. a une complexité $O(n^2)$ (où n = taille de l'entrée) »

↪ si n est assez grand, le nb. d'opérations est $\leq \text{constante} \times n^2$

- ▶ Avantages pour la théorie :

- ▶ Négliger les cas de bases
- ▶ Pas besoin de compter chaque opération en détail
- ▶ Flexibilité sur les opérations élémentaires

- ▶ Avantages pour la pratique :

- ▶ Indépendant des détails de programmation (nb. de variables intermédiaires, ...)
- ▶ Indépendant de l'environnement d'exécution : système d'exploitation, vitesse de la machine, compilateur, ...

Utilisation en complexité

« Mon algo. a une complexité $O(n^2)$ (où n = taille de l'entrée) »

↪ si n est assez grand, le nb. d'opérations est $\leq \text{constante} \times n^2$

► Avantages pour la théorie :

- Négliger les cas de bases
- Pas besoin de compter chaque opération en détail
- Flexibilité sur les opérations élémentaires

► Avantages pour la pratique :

- Indépendant des détails de programmation (nb. de variables intermédiaires, ...)
- Indépendant de l'environnement d'exécution : système d'exploitation, vitesse de la machine, compilateur, ...

Un temps de calcul dépend du moment et de l'endroit.

Un résultat de complexité reste vrai **pour toujours !**

Exemples

$$5n + 15 = O(n^2)$$

- ▶ Car pour $n \geq 8$, $5n + 15 \leq n^2$
- ▶ Ou alors pour $n \geq 3$, $5n + 15 \leq 5n^2$

$$\rightsquigarrow c = 1 \text{ et } n_0 = 8$$

$$\rightsquigarrow c = 5 \text{ et } n_0 = 3$$

Exemples

$$5n + 15 = O(n^2)$$

► Car pour $n \geq 8$, $5n + 15 \leq n^2$

$\rightsquigarrow c = 1$ et $n_0 = 8$

► Ou alors pour $n \geq 3$, $5n + 15 \leq 5n^2$

$\rightsquigarrow c = 5$ et $n_0 = 3$

1 <inst. 1>

2 **pour** $i = 1$ à n :

3 | <inst. 2>

4 **pour** $i = 1$ à n :

5 | **pour** $j = 1$ à n :

6 | | <inst. 3>

7 **renvoyer** *var*

► <inst. N> : opérations élémentaires

Exemples

$$5n + 15 = O(n^2)$$

► Car pour $n \geq 8$, $5n + 15 \leq n^2$

$\rightsquigarrow c = 1$ et $n_0 = 8$

► Ou alors pour $n \geq 3$, $5n + 15 \leq 5n^2$

$\rightsquigarrow c = 5$ et $n_0 = 3$

```
1 <inst. 1>
2 pour  $i = 1$  à  $n$  :
3   | <inst. 2>
4 pour  $i = 1$  à  $n$  :
5   |   pour  $j = 1$  à  $n$  :
6   |   | <inst. 3>
7 renvoyer  $var$ 
```

► <inst. N> : opérations élémentaires

► L1 et L7 : $O(1)$

► L2 exécute n fois L3 : $O(n)$

► L5 exécute n fois L6 : $O(n)$

► L4 exécute n fois L5 : $O(n^2)$

Total

$$2 \times O(1) + O(n) + O(n^2) = O(n^2)$$

Calcul avec les « grand O »

Lemme

- ▶ $O(f) + O(g) = O(f + g)$
- ▶ $O(f) \times O(g) = O(f \times g)$
- ▶ Si $f = O(g)$, alors $O(f + g) = O(g)$
- ▶ Si $\lambda \in \mathbb{R}_+$, $O(\lambda f) = O(f)$

Calcul avec les « grand O »

Lemme

- ▶ $O(f) + O(g) = O(f + g)$
- ▶ $O(f) \times O(g) = O(f \times g)$
- ▶ Si $f = O(g)$, alors $O(f + g) = O(g)$
- ▶ Si $\lambda \in \mathbb{R}_+$, $O(\lambda f) = O(f)$

Preuve du premier

- ▶ Soit $h_1 = O(f) : \exists c_1, n_1, \forall n \geq n_1, h_1(n) \leq c_1 f(n)$
- ▶ Soit $h_2 = O(g) : \exists c_2, n_2, \forall n \geq n_2, h_2(n) \leq c_2 g(n)$

Calcul avec les « grand O »

Lemme

- ▶ $O(f) + O(g) = O(f + g)$
- ▶ $O(f) \times O(g) = O(f \times g)$
- ▶ Si $f = O(g)$, alors $O(f + g) = O(g)$
- ▶ Si $\lambda \in \mathbb{R}_+$, $O(\lambda f) = O(f)$

Preuve du premier

- ▶ Soit $h_1 = O(f) : \exists c_1, n_1, \forall n \geq n_1, h_1(n) \leq c_1 f(n)$
- ▶ Soit $h_2 = O(g) : \exists c_2, n_2, \forall n \geq n_2, h_2(n) \leq c_2 g(n)$
- ▶ Donc $\forall n \geq \max(n_1, n_2)$,

$$\begin{aligned} h_1(n) + h_2(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq \max(c_1, c_2) f(n) + \max(c_1, c_2) g(n) \\ &\leq \max(c_1, c_2) (f(n) + g(n)) \end{aligned}$$

Calcul avec les « grand O »

Lemme

- ▶ $O(f) + O(g) = O(f + g)$
- ▶ $O(f) \times O(g) = O(f \times g)$
- ▶ Si $f = O(g)$, alors $O(f + g) = O(g)$
- ▶ Si $\lambda \in \mathbb{R}_+$, $O(\lambda f) = O(f)$

Preuve du premier

- ▶ Soit $h_1 = O(f) : \exists c_1, n_1, \forall n \geq n_1, h_1(n) \leq c_1 f(n)$
- ▶ Soit $h_2 = O(g) : \exists c_2, n_2, \forall n \geq n_2, h_2(n) \leq c_2 g(n)$
- ▶ Donc $\forall n \geq \max(n_1, n_2)$,

$$\begin{aligned} h_1(n) + h_2(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq \max(c_1, c_2) f(n) + \max(c_1, c_2) g(n) \\ &\leq \max(c_1, c_2) (f(n) + g(n)) \end{aligned}$$

$$\rightsquigarrow h_1 + h_2 = O(f + g)$$

« Grand O » et limites

Lemme

Soit $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$, et $c = \lim_{n \rightarrow +\infty} \left(\frac{f(n)}{g(n)} \right)$. Alors

- ▶ si $c = 0$: $f = O(g)$ mais $g \neq O(f)$
- ▶ si $0 < c < +\infty$: $f = O(g)$ et $g = O(f)$
- ▶ si $c = +\infty$: $f \neq O(g)$ mais $g = O(f)$

« Grand O » et limites

Lemme

Soit $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$, et $c = \lim_{n \rightarrow +\infty} \left(\frac{f(n)}{g(n)} \right)$. Alors

- ▶ si $c = 0$: $f = O(g)$ mais $g \neq O(f)$
- ▶ si $0 < c < +\infty$: $f = O(g)$ et $g = O(f)$
- ▶ si $c = +\infty$: $f \neq O(g)$ mais $g = O(f)$

Preuve

- ▶ Si $f(n)/g(n) \rightarrow_{\infty} c < +\infty$, alors $f(n)/g(n) \leq c + 1$ à partir d'un certain rang. Donc $f(n) \leq (c + 1)g(n)$, et $f = O(g)$.
- ▶ Si $f(n)/g(n) \rightarrow_{\infty} +\infty$, alors pour tout c , $f(n)/g(n) \geq c$ à partir d'un certain rang. Donc $f \neq O(g)$.

Inverse de limites

Lemme

Soit $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$, strictement positives pour n assez grand. Alors

- ▶ si $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}_+^*$, alors $\lim_{n \rightarrow +\infty} \frac{g(n)}{f(n)} = \frac{1}{c}$
- ▶ si $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = +\infty$, alors $\lim_{n \rightarrow +\infty} \frac{g(n)}{f(n)} = 0$
- ▶ si $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$, alors $\lim_{n \rightarrow +\infty} \frac{g(n)}{f(n)} = +\infty$

Preuve

Continuité de la fonction $x \mapsto \frac{1}{x}$ sur $]0, +\infty[$.

Inverse de limites

Lemme

Soit $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$, strictement positives pour n assez grand. Alors

- ▶ si $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}_+^*$, alors $\lim_{n \rightarrow +\infty} \frac{g(n)}{f(n)} = \frac{1}{c}$
- ▶ si $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = +\infty$, alors $\lim_{n \rightarrow +\infty} \frac{g(n)}{f(n)} = 0$
- ▶ si $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$, alors $\lim_{n \rightarrow +\infty} \frac{g(n)}{f(n)} = +\infty$

Preuve

Continuité de la fonction $x \mapsto \frac{1}{x}$ sur $]0, +\infty[$.

À retenir !

$$0 < c < +\infty$$

$\lim_{+\infty} f/g$	0	c	$+\infty$
$f = O(g)$	✓	✓	✗
$g = O(f)$	✗	✓	✓
$\lim_{+\infty} g/f$	$+\infty$	$1/c$	0

« Omega »

Définition

$f = \Omega(g)$ si (au choix !)

- ▶ $g = O(f)$
- ▶ $\exists c > 0, n_0 \geq 0, \forall n \geq n_0, f(n) \geq cg(n)$
- ▶ « pour n suffisamment grand, f est **supérieure** à g , à une constante multiplicative près »

« Omega »

Définition

$f = \Omega(g)$ si (au choix !)

- ▶ $g = O(f)$
- ▶ $\exists c > 0, n_0 \geq 0, \forall n \geq n_0, f(n) \geq cg(n)$
- ▶ « pour n suffisamment grand, f est **supérieure** à g , à une constante multiplicative près »

Remarque. Utilisé une seule fois dans le cours !

Les fonctions du cours

$\log n$, $\sqrt{n} = n^{1/2}$, n , n^2 , n^k , 2^n , $n!$

Les fonctions du cours

$$\log n, \quad \sqrt{n} = n^{1/2}, \quad n, \quad n^2, \quad n^k, \quad 2^n, \quad n!$$

Lemme de croissance comparée

Pour toutes **constantes** $\alpha, \beta > 0$,

$$\lim_{+\infty} \frac{(\log n)^\alpha}{n^\beta} = 0$$

$$\lim_{+\infty} \frac{n^\alpha}{(2^n)^\beta} = 0$$

$$\lim_{+\infty} \frac{(2^n)^\beta}{n!} = 0$$

(logarithmes \ll polynômes \ll exponentielles \ll factorielle)

Les fonctions du cours

$$\log n, \quad \sqrt{n} = n^{1/2}, \quad n, \quad n^2, \quad n^k, \quad 2^n, \quad n!$$

Lemme de croissance comparée

Pour toutes **constantes** $\alpha, \beta > 0$,

$$\lim_{+\infty} \frac{(\log n)^\alpha}{n^\beta} = 0$$

$$\lim_{+\infty} \frac{n^\alpha}{(2^n)^\beta} = 0$$

$$\lim_{+\infty} \frac{(2^n)^\beta}{n!} = 0$$

(logarithmes \ll polynômes \ll exponentielles \ll factorielle)

Si $f : \mathbb{N} \rightarrow \mathbb{R}_+$ tend vers $+\infty$ et $\alpha < \beta$:

$$\lim_{+\infty} \frac{f(n)^\alpha}{f(n)^\beta} = \lim_{+\infty} f(n)^{\alpha-\beta} = 0$$

Les fonctions du cours

$$\log n, \quad \sqrt{n} = n^{1/2}, \quad n, \quad n^2, \quad n^k, \quad 2^n, \quad n!$$

Lemme de croissance comparée

Pour toutes **constantes** $\alpha, \beta > 0$,

$$\lim_{+\infty} \frac{(\log n)^\alpha}{n^\beta} = 0$$

$$\lim_{+\infty} \frac{n^\alpha}{(2^n)^\beta} = 0$$

$$\lim_{+\infty} \frac{(2^n)^\beta}{n!} = 0$$

(logarithmes \ll polynômes \ll exponentielles \ll factorielle)

Si $f : \mathbb{N} \rightarrow \mathbb{R}_+$ tend vers $+\infty$ et $\alpha < \beta$:

$$\lim_{+\infty} \frac{f(n)^\alpha}{f(n)^\beta} = \lim_{+\infty} f(n)^{\alpha-\beta} = 0$$

Exemples

- ▶ $\log^2 n = O(\sqrt{n})$, $n^2 = O(n^3)$, ...
- ▶ Mais $\sqrt{n} \neq O(\log^2 n)$, $n^3 \neq O(n^2)$, ...

Calcul avec exponentielles et logarithmes

Sauf mention contraire : \log est le logarithme **en base 2**

$\rightsquigarrow \log n$ est environ le nombre de bits de n (exact : $\lfloor \log n \rfloor + 1$)

Calcul avec exponentielles et logarithmes

Sauf mention contraire : \log est le logarithme **en base 2**

$\rightsquigarrow \log n$ est environ le nombre de bits de n (exact : $\lfloor \log n \rfloor + 1$)

Règles du \log

- ▶ $\log 0$ non défini
- ▶ $\log 1 = 0$; $\log 2 = 1$
- ▶ $\log(a \times b) = \log a + \log b$
- ▶ $\log(a/b) = \log a - \log b$
- ▶ $\log(a^k) = k \log a$

Calcul avec exponentielles et logarithmes

Sauf mention contraire : \log est le logarithme **en base 2**

$\rightsquigarrow \log n$ est environ le nombre de bits de n (exact : $\lfloor \log n \rfloor + 1$)

Règles du \log

- ▶ $\log 0$ non défini
- ▶ $\log 1 = 0$; $\log 2 = 1$
- ▶ $\log(a \times b) = \log a + \log b$
- ▶ $\log(a/b) = \log a - \log b$
- ▶ $\log(a^k) = k \log a$

Règles de l'exponentielle

- ▶ $2^0 = 1$; $2^1 = 2$
- ▶ $2^{a+b} = 2^a \times 2^b$
- ▶ $2^{a-b} = 2^a / 2^b$
- ▶ $2^{a \times b} = (2^a)^b = (2^b)^a$
- ▶ $2^{\log a} = a = \log(2^a)$

Calcul avec exponentielles et logarithmes

Sauf mention contraire : \log est le logarithme **en base 2**

$\leadsto \log n$ est environ le nombre de bits de n (exact : $\lfloor \log n \rfloor + 1$)

Règles du \log

- ▶ $\log 0$ non défini
- ▶ $\log 1 = 0$; $\log 2 = 1$
- ▶ $\log(a \times b) = \log a + \log b$
- ▶ $\log(a/b) = \log a - \log b$
- ▶ $\log(a^k) = k \log a$

Règles de l'exponentielle

- ▶ $2^0 = 1$; $2^1 = 2$
- ▶ $2^{a+b} = 2^a \times 2^b$
- ▶ $2^{a-b} = 2^a / 2^b$
- ▶ $2^{a \times b} = (2^a)^b = (2^b)^a$
- ▶ $2^{\log a} = a = \log(2^a)$

« le \log . transforme les \times en $+$; l'exp. transforme les $+$ en \times »

Calcul avec exponentielles et logarithmes

Sauf mention contraire : log est le logarithme **en base 2**

↷ $\log n$ est environ le nombre de bits de n (exact : $\lfloor \log n \rfloor + 1$)

Règles du log

- ▶ $\log 0$ non défini
- ▶ $\log 1 = 0$; $\log 2 = 1$
- ▶ $\log(a \times b) = \log a + \log b$
- ▶ $\log(a/b) = \log a - \log b$
- ▶ $\log(a^k) = k \log a$

Règles de l'exponentielle

- ▶ $2^0 = 1$; $2^1 = 2$
- ▶ $2^{a+b} = 2^a \times 2^b$
- ▶ $2^{a-b} = 2^a / 2^b$
- ▶ $2^{a \times b} = (2^a)^b = (2^b)^a$
- ▶ $2^{\log a} = a = \log(2^a)$

« le log. transforme les \times en $+$; l'exp. transforme les $+$ en \times »

Exemples

$$2^{3n} = (2^3)^n = 8^n ; n^n = (2^{\log n})^n = 2^{n \log n}$$

Détermination d'un « grand O »

Objectif (souvent) : exprimer sous la forme $O(2^{cn^\alpha} n^\beta (\log n)^\gamma)$

Détermination d'un « grand O »

Objectif (souvent) : exprimer sous la forme $O(2^{cn^\alpha} n^\beta (\log n)^\gamma)$

Techniques usuelles

- ▶ Simplifier les constantes additives : $f(n + b) = O(f(n))$
- ▶ Supprimer les termes négligeables : $f + g = O(g)$ si $f = O(g)$
- ▶ Se ramener à des écritures *standard*. Exemples :
 - ▶ $\sqrt{n^3} / \sqrt[3]{n} = (n^3)^{1/2} / n^{1/3} = n^{3/2-1/3} = n^{7/6}$
 - ▶ $4^{\log n} = (2^{\log 4})^{\log n} = 2^{2 \log n} = (2^{\log n})^2 = n^2$
- ▶ Calculer des limites pour comparer des termes

Exemples de complexités de problèmes algorithmiques

Complexité	Notation O	Exemple d'algorithme
Constante	$O(1)$	Initialisation d'une variable
Logarithmique	$O(\log n)$	Recherche dichotomique dans un tableau trié
Linéaire	$O(n)$	Parcours d'un tableau (ou d'une liste)
Quasi-linéaire	$O(n \log n)$	Tri (fusion) d'un tableau
Quadratique	$O(n^2)$	Double boucle imbriquée
Cubique	$O(n^3)$	Triple boucle imbriquée
Exponentielle	$O(2^n)$	Énumération de tous les sous-ensembles de $\{1, \dots, n\}$
Factorielle	$O(n!)$	Énumération de toutes les permutations de $\{1, \dots, n\}$

Autres outils mathématiques

Factorielle et formule de Stirling

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 1 \sim_{n \rightarrow +\infty} \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

$\rightsquigarrow n! = O(\sqrt{n}(n/e)^n)$ par exemple

Autres outils mathématiques

Factorielle et formule de Stirling

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 1 \sim_{n \rightarrow +\infty} \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

$\rightsquigarrow n! = O(\sqrt{n}(n/e)^n)$ par exemple

Parties entières

- ▶ $\lfloor x \rfloor$ est le plus grand entier k tel que $k \leq x$
(et l'unique entier k tel que $k \leq x < k+1$)
- ▶ $\lceil x \rceil$ est le plus petit entier k tel que $x \leq k$
(et l'unique entier k tel que $k-1 < x \leq k$)
- ▶ $x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$
- ▶ Pour $n \in \mathbb{N}$, $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n$ (exercice !)

Sommes arithmétique et géométrique

$$\sum_{i=a}^b i = (b - a + 1) \cdot \frac{b + a}{2} \quad = \text{nb de termes} \times \text{moyenne}(\text{min}, \text{max})$$

$$\sum_{i=a}^b x^i = x^a \cdot \frac{x^{b-a+1} - 1}{x - 1} \quad = \frac{x^{b+1} - x^a}{x - 1}$$

Sommes arithmétique et géométrique

$$\sum_{i=a}^b i = (b - a + 1) \cdot \frac{b + a}{2}$$

= nb de termes \times moyenne(min, max)

$$\sum_{i=a}^b x^i = x^a \cdot \frac{x^{b-a+1} - 1}{x - 1}$$

$$= \frac{x^{b+1} - x^a}{x - 1}$$

Exemple :

```
S ← 0
pour i = 1 à n :
  y ← 1
  pour j = 1 à i :
    y ← x × y
  S ← S + y
renvoyer S
```

Sommes arithmétique et géométrique

$$\sum_{i=a}^b i = (b - a + 1) \cdot \frac{b + a}{2} \quad = \text{nb de termes} \times \text{moyenne}(\text{min}, \text{max})$$

$$\sum_{i=a}^b x^i = x^a \cdot \frac{x^{b-a+1} - 1}{x - 1} \quad = \frac{x^{b+1} - x^a}{x - 1}$$

Exemple :

```
S ← 0
pour i = 1 à n :
  y ← 1
  pour j = 1 à i :
    y ← x × y
  S ← S + y
renvoyer S
```

Complexité : $\sum_{i=1}^n i = \frac{n(n+1)}{2} = O(n^2)$

Sommes arithmétique et géométrique

$$\sum_{i=a}^b i = (b - a + 1) \cdot \frac{b + a}{2} \quad = \text{nb de termes} \times \text{moyenne}(\text{min}, \text{max})$$

$$\sum_{i=a}^b x^i = x^a \cdot \frac{x^{b-a+1} - 1}{x - 1} \quad = \frac{x^{b+1} - x^a}{x - 1}$$

Exemple :

```
S ← 0
pour i = 1 à n :
  y ← 1
  pour j = 1 à i :
    y ← x × y
  S ← S + y
renvoyer S
```

Complexité : $\sum_{i=1}^n i = \frac{n(n+1)}{2} = O(n^2)$

Valeur : $y = x^i \rightsquigarrow S = \sum_{i=1}^n x^i = \frac{x^{n+1} - x}{x - 1}$