Chapitre 1
Introduction

HLIN401 : Algorithmique et Complexité

Université de Montpellier
2018 – 2019

Déroulement du cours

- Cours : mardi 13h15-14h45, salle SC1.01
- TD/TP : par groupes (voir emploi du temps en ligne)
- Enseignants : B. Grenet (CM + TD/TP Gpe B), Julien Destombes (Gpe C), Nicolas Pompidor (Gpe A+CMI), Mohammed Senhaji (Gpe Math-Info)
- Évaluations : CC (type examen $\pm$ bonus/malus de TP) + Examen. Règle : $\max(Ex, 30\% CC + 70\% Ex)$
- **Ressources en ligne sur Moodle : cours HLIN401**

1. Exemple introductif : calculer x^n

But

- Pour un réel x et un entier $n \geq 1$, on veut calculer x^n
- Pour cela on va
  - proposer plusieurs algorithmes,
  - démontrer leur **validité**,
  - estimer leur **complexité**
    (= temps nécessaire au déroulement du programme)
  - voir une implémentation possible

But

- Pour un réel x et un entier $n \geq 1$, on veut calculer x^n
- Pour cela on va
  - proposer plusieurs algorithmes,
  - démontrer leur **validité**,
  - estimer leur **complexité**
    (= temps nécessaire au déroulement du programme)
  - voir une implémentation possible

Remarque. Problème très utile en pratique !

Algo 1 : multiplications successives

$\underline{\textsc{Algo1}(x, n)}$

y un réel;

$y \longleftarrow x$;

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

  $y \longleftarrow x * y$;

retourner y;

Algo 1 : multiplications successives

$\underline{\text{ALGO1}(x, n)}$

y un réel;

$y \longleftarrow x$;

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

$\quad y \longleftarrow x * y$;

retourner y;

Terminaison

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

Algo 1 : multiplications successives

$\underline{\textsc{Algo1}(x,n)}$

y un réel;

$y \longleftarrow x$;

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

  $y \longleftarrow x * y$;

retourner y;

Complexité (en temps)

Nombre d'*opérations élémentaires* :

- Déclaration de y ; affectation $(y \leftarrow x)$ $\rightsquigarrow$ **2 op.**
- Dans la boucle **pour** : incrémentation de i ; multiplication et affectation $(y \leftarrow x * y)$ $\rightsquigarrow$ **3 op.**
- $n-1$ répétitions de la boucle $\rightsquigarrow$ $\mathbf{3}\boldsymbol{n}\mathbf{-3}$ **op.**

$\rightsquigarrow$ **Complexité en temps** $\boldsymbol{O(n)}$

Algo 1 : multiplications successives

$\underline{\textsc{Algo1}(x, n)}$

y un réel;

$y \longleftarrow x$;

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

  $y \longleftarrow x * y$;

retourner y;

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

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

Algo 1 : multiplications successives

$\underline{\text{ALGO1}(x, n)}$

y un réel;

$y \longleftarrow x$;

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

$\quad y \longleftarrow x * y$;

retourner y;

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

$$\mathcal{P}_i \text{ : } \textit{après } i \textit{ tours de boucle, } y \textit{ contient } x^{i+1}$$

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 2 : Diviser pour régner

$\underline{\textsc{AlgoD\&C}(x,n)}$

si $n = 1$ **alors retourner** x;
sinon
  $z = \textsc{AlgoD\&C}(x, \lfloor n/2 \rfloor)$;
  si *n est pair* **alors retourner** $z \times z$;
  si *n est impair* **alors retourner** $x \times z \times z$;

Algo 2 : Diviser pour régner

$\underline{\textsc{AlgoD\&C}(x, n)}$

si $n = 1$ **alors retourner** x;

sinon

  $z = \textsc{AlgoD\&C}(x, \lfloor n/2 \rfloor)$;

  si n *est pair* **alors retourner** $z \times z$;

  si n *est impair* **alors retourner** $x \times z \times z$;

Terminaison

- **Nombre constant d'opérations** (≤ 5) + un appel récursif
- Appel récursif sur un **paramètre plus petit**
- **Cas de base** présent

$\rightsquigarrow$ L'algorithme termine.

Algo 2 : Diviser pour régner

$\underline{\text{ALGOD\&C}(x, n)}$

si $n = 1$ **alors retourner** x;
sinon
  $z = \text{ALGOD\&C}(x, \lfloor n/2 \rfloor)$;
  si *n est pair* **alors retourner** $z \times z$;
  si *n est impair* **alors retourner** $x \times z \times z$;

Complexité

Nombre constant d'opérations
$\leadsto$ complexité **proportionnelle au nombre d'appels récursifs**

Algo 2 : Diviser pour régner

$\underline{\textsc{AlgoD\&C}(x, n)}$

si $n = 1$ **alors retourner** x;
sinon
    $z = \textsc{AlgoD\&C}(x, \lfloor n/2 \rfloor)$;
    si *n est pair* **alors retourner** $z \times z$;
    si *n est impair* **alors retourner** $x \times z \times z$;

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

$$\mathcal{P}_n : \textsc{AlgoD\&C}(x, n) \textit{ fait au plus } \log n \textit{ appels récursifs}$$

Algo 2 : Diviser pour régner

$\underline{\text{ALGOD\&C}(x, n)}$

si $n = 1$ **alors retourner** x;
sinon
  $z = \text{ALGOD\&C}(x, \lfloor n/2 \rfloor)$;
  si *n est pair* **alors retourner** $z \times z$;
  si *n est impair* **alors retourner** $x \times z \times z$;

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

$\mathcal{P}_n$: $\text{ALGOD\&C}(x, n)$ *fait au plus* $\log n$ *appels récursifs*

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

$\rightsquigarrow$ **Complexité au plus proportionnelle à log** $\boldsymbol{n}$ (en $O(\log n)$).

Algo 2 : Diviser pour régner

$\underline{\textsc{AlgoD\&C}(x, n)}$

si $n = 1$ **alors retourner** x;
sinon
  $z = \textsc{AlgoD\&C}(x, \lfloor n/2 \rfloor)$;
  si *n est pair* **alors retourner** $z \times z$;
  si *n est impair* **alors retourner** $x \times z \times z$;

Validité : $\mathcal{P}_n$: $\textsc{AlgoD\&C}(x, n)$ *renvoie* x^n

Algo 2 : Diviser pour régner

$\underline{\textsc{AlgoD\&C}(x,n)}$

si $n = 1$ **alors retourner** x;
sinon
  $z = \textsc{AlgoD\&C}(x, \lfloor n/2 \rfloor)$;
  si *n est pair* **alors retourner** $z \times z$;
  si *n est impair* **alors retourner** $x \times z \times z$;

Validité : $\mathcal{P}_n$: $\textsc{AlgoD\&C}(x,n)$ *renvoie* x^n

- $n = 1$: $\textsc{AlgoD\&C}(x,1)$ renvoie $x \rightsquigarrow \mathcal{P}_1$ est vraie
- Soit $n \geq 2$ et supposons $\mathcal{P}_p$ pour tout $p < n$:
  $\textsc{AlgoD\&C}(x, \lfloor \frac{n}{2} \rfloor)$ renvoie $z = x^{\lfloor n/2 \rfloor}$ (car $\lfloor \frac{n}{2} \rfloor < n$!).
  - Si n est pair : $n = \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{2} \rfloor$ et $\textsc{AlgoD\&C}(x,n)$ renvoie $z \times z = x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} = x^n$.
  - Si n est impair, $n = 1 + \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{2} \rfloor$ et $\textsc{AlgoD\&C}(x,n)$ renvoie $x \times z \times z = x \times x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} = x^n$.

Donc $\mathcal{P}_n$ est vraie.

Algo 3 : Arnaque

$\textsc{AlgoArnaque}(x, n)$

retourner $pow(x, n)$;

Algo 3 : Arnaque

$\textsc{AlgoArnaque}(x, n)$

retourner $pow(x, n)$;

- Très pratique... mais qu'y a-t-il dessous ?
- Quelques idées :
  - `http://www.cplusplus.com/reference/cmath/pow/`
  - `https://www.quora.com/What-is-the-time-complexity-of-the-pow-function-in-c++-language-Is-it-log-b-or-O-1`
- Si on veut vraiment savoir, il faut analyser le code de *pow*...

2. Modèle pour la complexité algorithmique

Pourquoi un modèle ?

- Pour répondre à la question :
  Quel temps va nécessiter la résolution d'un problème algorithmique ?

Pourquoi un modèle ?

- Pour 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 ?

- Pour 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...
- Mais on va tout de même considérer un modèle, qui va nous permettre de faire des prédictions

Pourquoi un modèle ?

- Pour 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...
- Mais on va tout de même considérer un modèle, qui va nous permettre 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*.

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*.

Chaque **opération élémentaire** prend un **temps constant**

(modèle $\simeq$ *Word-RAM*)

Définition de la complexité

Dans ce modèle-là, on va :

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

Définition de la complexité

Dans ce modèle-là, on va :

- **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

Dans ce modèle-là, on va :

- **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

Dans ce modèle-là, on va :

- **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.

3. Conception et analyse d'un 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 !)
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 !)
3. Analyser l'algorithme :
   3.1 Terminaison

   3.2 Complexité en temps

   3.3 Validité 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 !)
3. Analyser l'algorithme :
   3.1 Terminaison
       - Souvent omise $\rightsquigarrow$ clair avec complexité et validité

   3.2 Complexité en temps

   3.3 Validité 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 !)
3. Analyser l'algorithme :
   3.1 Terminaison
      - Souvent omise $\rightsquigarrow$ clair avec complexité et validité

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

   3.3 Validité 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 !)
3. Analyser l'algorithme :
   3.1 Terminaison
      - Souvent omise $\rightsquigarrow$ clair avec complexité et validité

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

   3.3 Validité de l'algorithme
      - **Invariant** d'algorithme = propriété $\mathcal{P}_i$ valable après i tours de boucles / i appels récursifs.
      - Preuve par **récurrence**

Contenu du cours

0. Outils mathématiques pour l'analyse de complexité ($O(n)$, ...)

Contenu du cours

0. Outils mathématiques pour l'analyse de complexité ($O(n)$, ...)
1. Stratégies pour écrire des algorithmes (algorithmes gloutons, diviser pour régner, programmation dynamique, ...)

Contenu du cours

0. Outils mathématiques pour l'analyse de complexité ($O(n)$, ...)
1. Stratégies pour écrire des algorithmes (algorithmes gloutons, diviser pour régner, programmation dynamique, ...)
2. Structures de données :

Contenu du cours

0. Outils mathématiques pour l'analyse de complexité ($O(n)$, ...)
1. Stratégies pour écrire des algorithmes (algorithmes gloutons, diviser pour régner, programmation dynamique, ...)
2. Structures de données :
   - On utilisera celles que vous connaissez : types simples (entiers, réels, booléens, ...), chaînes de caractères, tableaux, listes (doublement) chaînées, piles, files, ...

Contenu du cours

0. Outils mathématiques pour l'analyse de complexité ($O(n)$, ...)
1. Stratégies pour écrire des algorithmes (algorithmes gloutons, diviser pour régner, programmation dynamique, ...)
2. Structures de données :
   - On utilisera celles que vous connaissez : types simples (entiers, réels, booléens, ...), chaînes de caractères, tableaux, listes (doublement) chaînées, piles, files, ...
   - On (re ?)verra deux autres : tas et arbres de recherche

Contenu du cours

0. Outils mathématiques pour l'analyse de complexité ($O(n)$, ...)
1. Stratégies pour écrire des algorithmes (algorithmes gloutons, diviser pour régner, programmation dynamique, ...)
2. Structures de données :
   - On utilisera celles que vous connaissez : types simples (entiers, réels, booléens, ...), chaînes de caractères, tableaux, listes (doublement) chaînées, piles, files, ...
   - On (re ?)verra deux autres : tas et arbres de recherche
3. Pour tous les algos, on utilisera la « recette » :

Contenu du cours

0. Outils mathématiques pour l'analyse de complexité ($O(n)$, ...)
1. Stratégies pour écrire des algorithmes (algorithmes gloutons, diviser pour régner, programmation dynamique, ...)
2. Structures de données :
   - On utilisera celles que vous connaissez : types simples (entiers, réels, booléens, ...), chaînes de caractères, tableaux, listes (doublement) chaînées, piles, files, ...
   - On (re ?)verra deux autres : tas et arbres de recherche
3. Pour tous les algos, on utilisera la « recette » :
   - Pseudo-code

Contenu du cours

0. Outils mathématiques pour l'analyse de complexité ($O(n)$, ...)
1. Stratégies pour écrire des algorithmes (algorithmes gloutons, diviser pour régner, programmation dynamique, ...)
2. Structures de données :
   - On utilisera celles que vous connaissez : types simples (entiers, réels, booléens, ...), chaînes de caractères, tableaux, listes (doublement) chaînées, piles, files, ...
   - On (re ?)verra deux autres : tas et arbres de recherche
3. Pour tous les algos, on utilisera la « recette » :
   - Pseudo-code
   - (Choix des structures de données)

Contenu du cours

0. Outils mathématiques pour l'analyse de complexité ($O(n)$, ...)
1. Stratégies pour écrire des algorithmes (algorithmes gloutons, diviser pour régner, programmation dynamique, ...)
2. Structures de données :
   - On utilisera celles que vous connaissez : types simples (entiers, réels, booléens, ...), chaînes de caractères, tableaux, listes (doublement) chaînées, piles, files, ...
   - On (re ?)verra deux autres : tas et arbres de recherche
3. Pour tous les algos, on utilisera la « recette » :
   - Pseudo-code
   - (Choix des structures de données)
   - Analyse : complexité et validité

Estimer la complexité en temps dans le pire cas de manière asymptotique : seule façon de faire ?

Estimer la complexité en temps dans le pire cas de manière asymptotique : seule façon de faire ?

Temps : d'autres mesures existent (espace mémoire, temps parallèle, ...)

Estimer la complexité en temps dans le pire cas de manière asymptotique : seule façon de faire ?

Temps : d'autres mesures existent (espace mémoire, temps parallèle, ...)

Pire cas : raffinements possibles (en moyenne, analyses amortie et lissée, cas pratiques, ...)

Estimer la complexité en temps dans le pire cas de manière asymptotique : seule façon de faire ?

Temps : d'autres mesures existent (espace mémoire, temps parallèle, ...)

Pire cas : raffinements possibles (en moyenne, analyses amortie et lissée, cas pratiques, ...)

Asymptotique : étude des « constantes cachées de le grand O », valeurs utiles en pratique, ...

Estimer la complexité en temps dans le pire cas de manière asymptotique : seule façon de faire ?

Temps : d'autres mesures existent (espace mémoire, temps parallèle, ...)

Pire cas : raffinements possibles (en moyenne, analyses amortie et lissée, cas pratiques, ...)

Asymptotique : étude des « constantes cachées de le grand O », valeurs utiles en pratique, ...

Dans ce cours, on choisit ce modèle de complexité car c'est **le plus simple**

Estimer la complexité en temps dans le pire cas de manière asymptotique : seule façon de faire ?

Temps : d'autres mesures existent (espace mémoire, temps parallèle, ...)

Pire cas : raffinements possibles (en moyenne, analyses amortie et lissée, cas pratiques, ...)

Asymptotique : étude des « constantes cachées de le grand O », valeurs utiles en pratique, ...

Dans ce cours, on choisit ce modèle de complexité car c'est **le plus simple** et :

- souvent suffisant
- on commence toujours par ça
- si on comprend comment marche ce modèle, on saura (plus tard !) comprendre les autres