

- Exemple introductif et modèle de calcul -

HLIN508: Fondement de l'algorithmique

2018

-I- Exemple introductif -

- Comment calculer x^n -

But

- ▶ Pour un réel x et un entier $n \geq 1$, on veut calculer x^n , sur une machine.
- ▶ Pour cela on va proposer plusieurs algorithmes, démontrer leur **validité** et estimer leur **complexité**, c'est-à-dire les ressources en temps et en espace mémoire nécessaire au déroulement du programme.

On verra aussi leur implémentation...

Algo 1: Tableau

ALGOTABLEAU (x, n)

T un tableau de taille n ;

$T[0] \leftarrow x$;

pour tous les i de 1 à $n - 1$ faire

$T[i] \leftarrow x * T[i - 1]$;

retourner $T[n - 1]$;

Analyse de l'algo:

Algo 1: Tableau

ALGOTABLEAU (x, n)

T un tableau de taille n ;

$T[0] \leftarrow x$;

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

$T[i] \leftarrow x * T[i - 1]$;

retourner $T[n - 1]$;

Analyse de l'algo:

► **Terminaison:**

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

Algo 1: Tableau

ALGOTABLEAU (x, n)

T un tableau de taille n ;

$T[0] \leftarrow x$;

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

$T[i] \leftarrow x * T[i - 1]$;

retourner $T[n - 1]$;

Analyse de l'algo:

► **Terminaison:**

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

► **Complexité** (en espace):

On utilise 3 variables (x , n et i) et un tableau de taille n .

En dehors des besoins du système, on consomme $n + 3$ cases mémoire.

On dira qu'on a une **complexité en espace en** $O(n)$.

Algo 1: Tableau

ALGO TABLEAU (x, n)

T un tableau de taille n ;

$T[0] \leftarrow x$;

pour tous les i de 1 à $n - 1$ faire

$T[i] \leftarrow x * T[i - 1]$;

retourner $T[n - 1]$;

Analyse de l'algo:

► **Complexité** (en temps):

On compte le nombre *d'opérations élémentaires*:

- Hors de la boucle **pour**, on déclare un tableau, on affecte une valeur et on retourne une valeur: **3 op.**
- Dans la boucle **pour**, on incrémente une variable (i), on fait une multiplication, et on affecte une variable ($T[i]$):
3 op. faites $n - 1$ fois: **$3n - 3$ op.**

En tout on fait **$3n$ opérations élémentaires**.

On dira qu'on a une **complexité en temps en $O(n)$** .

Algo 1: Tableau

ALGOTABLEAU (x, n)

T un tableau de taille n ;

$T[0] \leftarrow x$;

pour tous les i de 1 à $n - 1$ faire

$T[i] \leftarrow x * T[i - 1]$;

retourner $T[n - 1]$;

Analyse de l'algo:

► Validité :

On va établir la propriété (**invariant de l'algorithme**):

$\mathcal{P}_i$: 'après i tours de boucle, $T[i]$ contient x^{i+1} '

Pour cela, on procède par **récurrence** (l'arme fatale!):

- Pour $i = 0$, $\mathcal{P}_0$ est vraie: avant la boucle, $T[0]$ vaut x ($= x^1$).
- Supposons que pour $i \geq 1$, $\mathcal{P}_{i-1}$ soit vraie. Alors au i ème tour de boucle, $T[i]$ prendra la valeur $x \times T[i - 1]$. Comme on a déjà fait $i - 1$ tour de boucles et que $\mathcal{P}_{i-1}$ est vrai, $T[i - 1]$ vaut $x^{(i-1)+1} = x^i$. Donc $T[i] = x \times x^i = x^{i+1}$ et $\mathcal{P}_i$ est vraie.

Donc, par récurrence, $T[n - 1] = x^n$ à la fin de l'algo.

Algo 2: Sans tableau

ALGO SANS TABLEAU (x, n)

y un réel;

$y \leftarrow x$;

pour tous les i de 1 à $n - 1$ faire

$y \leftarrow x * y$;

retourner y ;

Analyse de l'algo:

Algo 2: Sans tableau

ALGO SANS TABLEAU (x, n)

y un réel;

$y \leftarrow x$;

pour tous les i de 1 à $n - 1$ faire

$y \leftarrow x * y$;

retourner y ;

Analyse de l'algo:

► **Terminaison:**

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

Algo 2: Sans tableau

ALGO SANS TABLEAU (x, n)

y un réel;

$y \leftarrow x$;

pour tous les i de 1 à $n - 1$ faire

$y \leftarrow x * y$;

retourner y ;

Analyse de l'algo:

► **Terminaison:**

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

► **Complexité** (en espace):

On utilise 4 variables (x , n , y et i). En dehors des besoins du système, on consomme 4 cases mémoire.

On dira qu'on a une **complexité en espace en $O(1)$** .

Algo 2: Sans tableau

ALGO SANS TABLEAU (x, n)

y un réel;

$y \leftarrow x$;

pour tous les i de 1 à $n - 1$ faire

$y \leftarrow x * y$;

retourner y ;

Analyse de l'algo:

- **Complexité** (en temps):

On compte le nombre *d'opérations élémentaires*:

- Hors de la boucle **pour**, on déclare une variable (y), on affecte une valeur et on retourne une valeur: **3 op.**
- Dans la boucle **pour**, on incrémente une variable (i), on fait une multiplication, et on affecte une variable (y):
3 op. faites $n - 1$ fois: **$3n - 3$ op.**

En tout on fait **$3n$ opérations élémentaires**.

On dira qu'on a une **complexité en temps en $O(n)$** .

Algo 2: Sans tableau

ALGO SANS TABLEAU (x, n)

y un réel;

$y \leftarrow x$;

pour tous les i de 1 à $n - 1$ faire

$y \leftarrow x * y$;

retourner y ;

Analyse de l'algo:

► Validité :

On va établir la propriété (**invariant de l'algorithme**):

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

Pour cela, on procède par **récurrence** (quelle surprise...):

- Pour $i = 0$, $\mathcal{P}_0$ est vraie: avant la boucle, y vaut x ($= x^1$).
- Supposons que pour $i \geq 1$, $\mathcal{P}_{i-1}$ soit vraie. Alors au i ème tour de boucle, y prendra la valeur $x \times y$. Comme on a déjà fait $i - 1$ tour de boucles et que $\mathcal{P}_{i-1}$ est vrai, y vaut $x^{(i-1)+1} = x^i$ à ce moment. Donc y prend la valeur $x \times y = x^{i+1}$ et $\mathcal{P}_i$ est vraie.

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

Algo 3: Dichotomie ('divide and conquer' !)

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** $x \times z \times z$;

Analyse de l'algo:

Algo 3: Dichotomie ('divide and conquer' !)

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** $x \times z \times z$;

Analyse de l'algo:

► **Terminaison:**

On fait un nombre fixe d'opération et un appel récursif. À chacun de ces appels le second paramètre diminue strictement, donc l'algo termine.

Algo 3: Dichotomie ('divide and conquer' !)

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** $x \times z \times z$;

Analyse de l'algo:

- **Terminaison:**

On fait un nombre fixe d'opération et un appel récursif. À chacun de ces appels le second paramètre diminue strictement, donc l'algo termine.

- **Complexité** (en temps et en espace):

A chaque appel, l'algorithme fait un nombre borné d'opérations élémentaires (au pire 3 tests, 3 multiplications et 1 retour de valeur). *La complexité en temps et en espace est proportionnelle au nombre d'appel récursifs.*

Algo 3: Dichotomie ('divide and conquer' !)

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** $x \times z \times z$;

Analyse de l'algo:

- **Nbre d'appels récursifs** (nbre de fois qu'on peut diviser n par 2 $\rightsquigarrow \log n$)

Par **réurrence** ('forte' cette fois...):

$\mathcal{P}_p$: 'ALGOD&C(x, p) fait au plus $\log p$ appels récursifs'

- Vrai pour $p = 1$...
- Pour $n \geq 2$ supposons que $\mathcal{P}_p$ soit vrai pour tout $p < n$. Le nombre d'appels réc. de ALGOD&C(x, n) est 1+ le nombre d'appels réc. de ALGOD&C($x, \lfloor n/2 \rfloor$). Par $\mathcal{P}_{\lfloor n/2 \rfloor}$, c'est $\leq 1 + \log \lfloor n/2 \rfloor \leq 1 + \log(n/2) = 1 + (\log n - 1) = \log n$

Complexité (en tps et esp) au plus proportionnelle à $\log n$ (en $O(\log n)$).

Algo 3: Dichotomie ('divide and conquer' !)

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** $x \times z \times z$;

Analyse de l'algo:

► Validité :

On va établir $\mathcal{P}_p$: ' $\text{ALGOD\&C}(x, p)$ renvoie x^p ' (réc...)

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

► Pour $n \geq 2$ supposons que $\mathcal{P}_p$ soit vrai pour tout $p < n$.

Comme $\lfloor n/2 \rfloor < n$, $\mathcal{P}_{\lfloor n/2 \rfloor}$ est vraie et $\text{ALGOD\&C}(x, \lfloor n/2 \rfloor)$ renvoie $x^{\lfloor n/2 \rfloor}$.

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

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

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

Algo 4: Algo de l'arnaque

ALGOARNAQUE (x, n)

retourner $pow(x, n)$;

Algo 4: Algo de l'arnaque

ALGOARNAQUE (x, n)

retourner $\text{pow}(x, n)$;

- ▶ Très pratique, mais qu'est ce qu'il y a dessous...
- ▶ Quelques idées:

<http://www.cplusplus.com/reference/cmath/pow/>

[https://www.quora.com/](https://www.quora.com/What-is-the-time-complexity-of-the-pow-function-in-c++-language-Is-it-log-b-or-0-1)

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*...

-II- Modèle pour la complexité algorithmique -

Modèle choisi

- Pour répondre à la question:
Quelle ressource (en temps et en espace) va nécessiter la résolution d'un problème algorithmique?

Modèle choisi

- ▶ Pour répondre à la question:
Quelle ressource (en temps et en espace) va nécessiter la résolution d'un problème algorithmique?
- ▶ Difficile à estimer: dépend du programme, du langage, de la machine, du système d'exploitation...

Modèle choisi

- ▶ Pour répondre à la question:
Quelle ressource (en temps et en espace) 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...

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 **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 **boucles**: *pour* et *tant que*.
- ▶ Dans notre modèle, on va considérer que:
 - Chaque **opération élémentaire** prend un **temps constant**
 - Chaque **déclaration de variable (simple)** et appel de fonction consomme un **espace machine 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 **boucles**: *pour* et *tant que*.
- ▶ Dans notre modèle, on va considérer que:
 - Chaque **opération élémentaire** prend un **temps constant**
 - Chaque **déclaration de variable (simple)** et **appel de fonction** consomme un **espace machine constant**.

(modèle *WORD-RAM*, pas très vrai en fait, mais suffisant ici...)

Modèle choisi

Dans ce modèle-là, on va:

- ▶ **compter le nombre d'opérations élémentaires** (pour établir la **complexité en temps**)
- ▶ **compter le nombre de déclarations de variables et d'appels de fonctions** (pour établir la **complexité en espace**)

Modèle choisi

Dans ce modèle-là, on va:

- ▶ **compter le nombre d'opérations élémentaires** (pour établir la **complexité en temps**)
- ▶ **compter le nombre de déclarations de variables et d'appels de fonctions** (pour établir la **complexité en espace**)
- ▶ Exprimer ces valeurs **en fonction des paramètres d'entrée** de l'algorithme.

Modèle choisi

Dans ce modèle-là, on va:

- ▶ **compter le nombre d'opérations élémentaires** (pour établir la **complexité en temps**)
- ▶ **compter le nombre de déclarations de variables et d'appels de fonctions** (pour établir la **complexité en espace**)
- ▶ Exprimer ces valeurs **en fonction des paramètres d'entrée** de l'algorithme.
- ▶ De manière asymptotique

Modèle choisi

Dans ce modèle-là, on va:

- ▶ **compter le nombre d'opérations élémentaires** (pour établir la **complexité en temps**)
- ▶ **compter le nombre de déclarations de variables et d'appels de fonctions** (pour établir la **complexité en espace**)
- ▶ 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.

-III- Conception et analyse d'un algorithme -

Conception et analyse d'un algorithme

La 'recette' pour cela:

1. Ecrire le pseudo-code de l'algorithme
2. Choisir les structures de données à utiliser les variables (va influencer la complexité de l'algo!)
3. Analyser l'algorithme:

Conception et analyse d'un algorithme

La 'recette' pour cela:

1. Ecrire le pseudo-code de l'algorithme
2. Choisir les structures de données à utiliser les variables (va influencer la complexité de l'algo!)
3. Analyser l'algorithme:
 - 3.1 Terminaison
 - 3.2 Complexité en temps et en espace
 - 3.3 Validité de l'algorithme

Conception et analyse d'un algorithme

Quelques remarques:

1. Dans ce cours, on va voir certaines stratégies pour écrire des algorithmes...

Conception et analyse d'un algorithme

Quelques remarques:

1. Dans ce cours, on va voir certaines stratégies pour écrire des algorithmes...
2. Vous connaissez certaines structures de données: les types simples (entiers, réels, booléens, caractères...), les chaînes de caractères, les tableaux, les listes (doublement) chaînées, les piles, les files...
On va en (re?)voir deux autres: les tas et les arbres de recherche.

Conception et analyse d'un algorithme

Quelques remarques:

1. Dans ce cours, on va voir certaines stratégies pour écrire des algorithmes...
2. Vous connaissez certaines structures de données: les types simples (entiers, réels, booléens, caractères...), les chaînes de caractères, les tableaux, les listes (doublement) chaînées, les piles, les files...
On va en (re?)voir deux autres: les tas et les arbres de recherche.
3. Sur la validité de l'algorithme:

- ▶ Parfois (souvent), on omet la terminaison, cela découle de la borne sur la complexité en temps.
- ▶ On cherche une borne supérieure sur la complexité de l'algorithme dans le pire des cas ('je suis sûr que mon algo ne consommera pas plus que...')
- ▶ Pour prouver la validité de l'algorithme, on utilise souvent un **invariant d'algorithme**, une propriété vraie tout au long de l'algorithme (du genre $\mathcal{P}_i$: 'après i tour de boucles, appels récursifs, on a ...').

On montre souvent la validité de l'invariant par **récence** (sur le nombre de tour de boucles, d'appels récursifs...)