DIU Enseignement de l'Info au Lycée

Cours 1 : Algorithmes, modèle de calcul, complexité

Université de Montpellier
2018 – 2019

1. Exemple introductif : calculer x^n
2. Un algorithme ?
3. Modèle pour la complexité algorithmique
4. Conception et analyse d'un algorithme
5. 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 **validité**,
  - estimer leur **complexité**
    ($=$ **temps** et **espace mémoire** nécessaires 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 **validité**,
  - estimer leur **complexité**
    (= **temps** et **espace mémoire** nécessaires au déroulement du programme)
  - voir une implémentation possible

Remarque : Problème très utile en pratique ! (Résolution d'équa. diff., Crypto : RSA, courbes elliptiques...)

Algo 1 : Multiplications successives avec tableau

AlgoTableau (x, n)

T un tableau de taille n;

$T[0] \longleftarrow x$;

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

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

retourner $T[n-1]$;

Algo 1 : Multiplications successives avec tableau

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

T un tableau de taille n;
$T[0] \longleftarrow x$;
pour tous les *i de 1 à* $n-1$ **faire**
  $T[i] \longleftarrow x * T[i-1]$;
retourner $T[n-1]$;

Un petit exemple :

On effectue **l'appel** ALGOTABLEAU $(3, 5)$:

- Initialisation de T : $T=$

3				

- Étape $i = 1$: $T=$

3	9			

- Étape $i = 2$: $T=$

3	9	27		

- Étape $i = 3$: $T=$

3	9	27	81	

- Étape $i = 4$: $T=$

3	9	27	81	243

- **L'algo retourne 243**

Algo 1 : Multiplications successives avec tableau

AlgoTableau (x, n)

T un tableau de taille n;
$T[0] \longleftarrow x$;
pour tous les i *de 1 à* $n-1$ **faire**
  $T[i] \longleftarrow x * T[i-1]$;
retourner $T[n-1]$;

Terminaison :

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

Complexité (en espace) :

Nombre de *déclarations élémentaires* :

- Récupérations des paramètres : x et n $\rightsquigarrow$ **2 cases mém.**
- Déclaration de T ; $\rightsquigarrow$ **n cases mém.**
- Déclaration de i ; $\rightsquigarrow$ **1 case mém.**

En tout $n+3$ cases mém. $\rightsquigarrow$ **Complexité en espace $O(n)$**

Algo 1 : Multiplications successives avec tableau

```
ALGOTABLEAU (x, n)
T un tableau de taille n;
T[0] ⟵ x;
pour tous les i de 1 à n − 1 faire
    T[i] ⟵ x ∗ T[i − 1];
retourner T[n − 1];
```

Complexité (en temps) :

Nombre *d'opérations élémentaires* :

- Hors du '**pour**' : récupération des param., déclaration de T, affectation de $T[0]$ et retour de $T[n-1]$: $\rightsquigarrow$ **5 op.**
- Dans le '**pour**', récup. de $T[i-1]$, incrément de i, une mult., affectation de $T[i]$: 4 op. faites $n-1$ fois : $\rightsquigarrow$ $\mathbf{4}\boldsymbol{n} - \mathbf{4}$ **op.**

En tout $4n+1$ op. élém. $\rightsquigarrow$ **Complexité en temps** $\boldsymbol{O(n)}$

Algo 1 : Multiplications successives avec tableau

```
AlgoTableau (x, n)
T un tableau de taille n;
T[0] ⟵ x;
pour tous les i de 1 à n − 1 faire
    T[i] ⟵ x ∗ T[i − 1];
retourner T[n − 1];
```

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

$$\mathcal{P}_i : \text{"après } i \text{ tours de boucle, } T[i] \text{ contient } x^{i+1} \text{"}$$

Preuve par **récurrence** (l'arme fatale !) :

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

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

Algo 1 : Multiplications successives avec tableau

<u>ALGOTABLEAU (x, n)</u>

T un tableau de taille n;

$T[0] \longleftarrow x$;

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

$\quad T[i] \longleftarrow x * T[i-1]$;

retourner $T[n-1]$;

Exemple de code :

AlgoTableau.py ▸ ...

```python
def AlgoTableau(x,n):
    T=[]
    T.append(x)
    for i in range(1, n):
        T.append(T[i-1]*x)
    return T[n-1]

print(AlgoTableau(2.5,8))
```

Algo 2 : Multiplications successives sans tableau

AlgoSansTableau(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 2 : Multiplications successives sans tableau

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

y un réel;

$y \longleftarrow x$;

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

    $y \longleftarrow x * y$;

retourner y;

Un petit exemple :

On effectue **l'appel** AlgoSansTableau $(2, 5)$:

- Initialisation de y : $y = 2$
- Étape $i = 1$: $y = 2 * 2 = 4$
- Étape $i = 2$: $y = 2 * 4 = 8$
- Étape $i = 3$: $y = 2 * 8 = 16$
- Étape $i = 4$: $y = 2 * 16 = 32$

- **L'algo retourne 32**

Algo 2 : Multiplications successives sans tableau

AlgoSansTableau(x, n)

y un réel;
$y \longleftarrow x$;
pour tous les i *de 1 à* $n - 1$ **faire**
    $y \longleftarrow x * y$;
retourner y;

Terminaison

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

Complexité (en espace) :

Nombre de *déclarations élémentaires* :

- Récupérations des paramètres : x et n $\rightsquigarrow$ **2 cases mém.**
- Déclaration de y ; $\rightsquigarrow$ **1 cases mém.**

En tout 3 cases mém. $\rightsquigarrow$ **Complexité en espace $O(1)$**

Algo 2 : Multiplications successives sans tableau

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

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 $\rightsquigarrow$ **3n − 3 op.**

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

Algo 2 : Multiplications successives sans tableau

```
AlgoSansTableau(x, n)
y un réel;
y ⟵ x;
pour tous les i de 1 à n − 1 faire
    y ⟵ 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}* "

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 : Multiplications successives sans tableau

<u>ALGOSANSTABLEAU(x, n)</u>

y un réel;

$y \longleftarrow x$;

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

  $y \longleftarrow x * y$;

retourner y;

Exemple de code :

AlgoSansTableau.py ▸ ...

```python
def AlgoSansTableau(x,n):
    y=x
    for i in range(1, n):
        y=y*x
    return y

print(AlgoSansTableau(2.5,8))
```

Algo 3 : 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 3 : 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** $x \times z \times z$;

Un petit exemple :

On effectue **l'appel** $\textsc{AlgoD\&C}$ $(3,5)$:

- $\textsc{AlgoD\&C}$ $(3,5)$ calcule $z = \textsc{AlgoD\&C}$ $(3,2)$ et retourne $3.z^2$
- $\textsc{AlgoD\&C}$ $(3,2)$ calcule $z = \textsc{AlgoD\&C}$ $(3,1)$ et retourne z^2
- $\textsc{AlgoD\&C}$ $(3,1)$ retourne 3
- Du coup, $\textsc{AlgoD\&C}$ $(3,2)$ retourne $3^2 = 9$

- Du coup, $\textsc{AlgoD\&C}$ $(3,5)$ **retourne** $3.9^2 = 243$

Algo 3 : Diviser pour régner

```
ALGOD&C(x, n)
si n = 1 alors retourner x;
sinon
    z = ALGOD&C(x, ⌊n/2⌋);
    si n est pair alors retourner z × z;
    si n est impair alors retourner x × z × z;
```

Terminaison

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

$\rightsquigarrow$ L'algorithme termine.

Algo 3 : 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 (en espace et en temps)

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

Algo 3 : 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 $\rightsquigarrow \log n$)

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

Algo 3 : Diviser pour régner

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

si $n = 1$ **alors retourner** x;
sinon
  $z = $ 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 $\rightsquigarrow \log n$)

$\mathcal{P}_n$: "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 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 $\boldsymbol{n}$** (en $O(\log n)$)

Algo 3 : 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 3 : Diviser pour régner

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

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

Exemple de code :

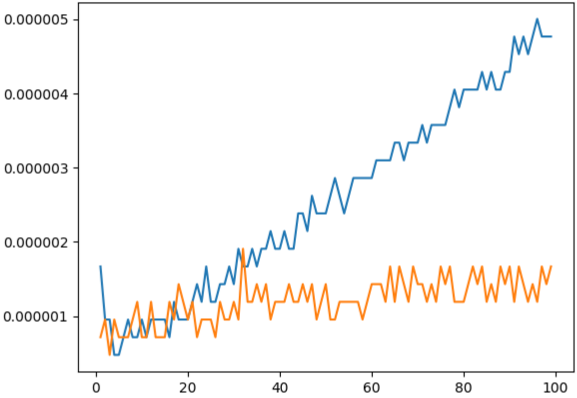
AlgoDetC.py ▸ ...

```python
def AlgoDetC(x,n):
    if n==1:
        return x
    else :
        z=AlgoDetC(x,n//2)
        if(n%2==0):
            return z*z
        else:
            return x*z*z


print(AlgoDetC(2.5,8))
```

Comparaison : algo itératif Vs algo récursif

On compare les temps mis par les appels $\textsc{AlgoSansTableau}$ $(2, n)$ et $\textsc{AlgoD\&C}(2, n)$ pour $n = 1, \ldots, 100$.



Remarque : En effet, c'est crédible que '$O(n)$ croît plus vite que $O(\log n)$' ...

Comparaison : algo itératif Vs algo récursif : Le Code

ComparaisonIteratifVsDC.py › AlgoDetC

```python
import time
import matplotlib.pyplot as plt

def AlgoSansTableau(x,n):
    y=x
    for i in range(1, n):
        y=y*x
    return y

def AlgoDetC(x,n):
    if n==1:
        return x
    else :
        z=AlgoDetC(x,n//2)
        if(n%2==0):
            return z*z
        else:
            return x*z*z

#Valeurs de n choisies
abscisses = [n for n in range(1,100)]
#Temps de calcul
tps1 = []
for n in range(1,100):
    t=time.time()
    AlgoSansTableau(2,n)
    tps1.append(time.time()-t)
tps2 = []
for n in range(1,100):
    t=time.time()
    AlgoDetC(2,n)
    tps2.append(time.time()-t)
#Tracé
plt.plot(abscisses, tps1)
plt.plot(abscisses, tps2)
plt.show()
```

Algo 3 : Arnaque

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

retourner $x ** n$;

Algo 3 : Arnaque

ALGOARNAQUE(x, n)

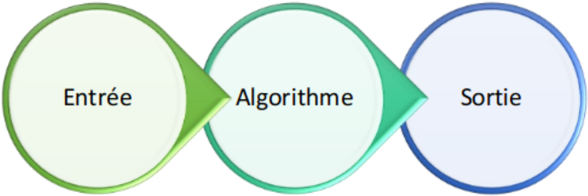
retourner $x * *n$;

- Très pratique... mais qu'y a-t-il dessous ?
- Une discussion :
  - https://stackoverflow.com/questions/1019740/speed-of-calculating-powers-in-python
- Si on veut vraiment savoir, il faut analyser le code source Python de $**$...

Définition d'un algorithme

- Un algorithme est une procédure pas-à-pas de résolution d'un problème.

- Généralement, l'algorithme prend en paramètres des **valeurs d'entrée** et produit des **valeurs de sortie** :



[1]

- L'analogie classique : **la recette de cuisine !**

1. Image emprûntée à nos collègues lyonnais, préparateurs du DIU...

Le véritable aïoli montpellierain !

L'aïoli montpellierain :
Aïoli pour 4 personnes :
3 œufs, 2 gousses d'ail,
400ml d'huile d'olive, du sel

1. Prendre un bol
2. Pour chaque œuf, séparer le blanc du jaune, verser le jaune dans le bol
3. Rajouter les deux gousses d'ail écrasées
4. Tant qu'il reste de l'huile, l'ajouter petit-à-petit en remuant
5. Si c'est fade, ajouter du sel

Le véritable aïoli montpellierain !

L'aïoli montpellierain :
Aïoli pour 4 personnes :
3 œufs, 2 gousses d'ail,
400ml d'huile d'olive, du sel

1. Prendre un bol
2. Pour chaque œuf, séparer le blanc du jaune, verser le jaune dans le bol
3. Rajouter les deux gousses d'ail écrasées
4. Tant qu'il reste de l'huile, l'ajouter petit-à-petit en remuant
5. Si c'est fade, ajouter du sel

- **Un nom :**
  L'aïoli montpellierain

Le véritable aïoli montpellierain !

L'aïoli montpellierain :
Aïoli pour 4 personnes :
3 œufs, 2 gousses d'ail,
400ml d'huile d'olive, du sel

1. Prendre un bol
2. Pour chaque œuf, séparer le blanc du jaune, verser le jaune dans le bol
3. Rajouter les deux gousses d'ail écrasées
4. Tant qu'il reste de l'huile, l'ajouter petit-à-petit en remuant
5. Si c'est fade, ajouter du sel

- **Des paramètres d'entrée :**
  3 œufs, 2 gousses d'ail,
  400ml d'huile d'olive, du sel

Le véritable aïoli montpellierain !

L'aïoli montpellierain :
Aïoli pour 4 personnes :
3 œufs, 2 gousses d'ail,
400ml d'huile d'olive, du sel

1. Prendre un bol
2. Pour chaque œuf, séparer le blanc du jaune, verser le jaune dans le bol
3. Rajouter les deux gousses d'ail écrasées
4. Tant qu'il reste de l'huile, l'ajouter petit-à-petit en remuant
5. Si c'est fade, ajouter du sel

- **Une valeur de sortie :**
  Aïoli pour 4 personnes

Le véritable aïoli montpellierain !

L'aïoli montpellierain :
Aïoli pour 4 personnes :
3 œufs, 2 gousses d'ail,
400ml d'huile d'olive, du sel

1. Prendre un bol
2. Pour chaque œuf, séparer le blanc du jaune, verser le jaune dans le bol
3. Rajouter les deux gousses d'ail écrasées
4. Tant qu'il reste de l'huile, l'ajouter petit-à-petit en remuant
5. Si c'est fade, ajouter du sel

- **Une déclaration de variable :**
  1. Prendre un bol

Le véritable aïoli montpellierain !

L'aïoli montpellierain :

Aïoli pour 4 personnes :

3 œufs, 2 gousses d'ail,
400ml d'huile d'olive, du sel

1. Prendre un bol
2. Pour chaque œuf, séparer le blanc du jaune, verser le jaune dans le bol
3. Rajouter les deux gousses d'ail écrasées
4. Tant qu'il reste de l'huile, l'ajouter petit-à-petit en remuant
5. Si c'est fade, ajouter du sel

- **Une instruction élémentaire :**

  3. Rajouter les deux gousses d'ail écrasées

Le véritable aïoli montpellierain !

L'aïoli montpellierain :
Aïoli pour 4 personnes :
3 œufs, 2 gousses d'ail,
400ml d'huile d'olive, du sel

1. Prendre un bol
2. Pour chaque œuf, séparer le blanc du jaune, verser le jaune dans le bol
3. Rajouter les deux gousses d'ail écrasées
4. Tant qu'il reste de l'huile, l'ajouter petit-à-petit en remuant
5. Si c'est fade, ajouter du sel

- **Un test** *('Si ... alors ...')* **:**
  5. Si c'est fade, ajouter du sel

Le véritable aïoli montpellierain !

L'aïoli montpellierain :
Aïoli pour 4 personnes :
3 œufs, 2 gousses d'ail,
400ml d'huile d'olive, du sel

1. Prendre un bol
2. Pour chaque œuf, séparer le blanc du jaune, verser le jaune dans le bol
3. Rajouter les deux gousses d'ail écrasées
4. Tant qu'il reste de l'huile, l'ajouter petit-à-petit en remuant
5. Si c'est fade, ajouter du sel

- **Une boucle itérative** *('Pour ... faire ...')* **:**

  2. Pour chaque œuf, séparer le blanc du jaune, verser le jaune dans le bol

Le véritable aïoli montpellierain !

L'aïoli montpellierain :
Aïoli pour 4 personnes :
3 œufs, 2 gousses d'ail,
400ml d'huile d'olive, du sel

1. Prendre un bol
2. Pour chaque œuf, séparer le blanc du jaune, verser le jaune dans le bol
3. Rajouter les deux gousses d'ail écrasées
4. Tant qu'il reste de l'huile, l'ajouter petit-à-petit en remuant
5. Si c'est fade, ajouter du sel

- **Une boucle conditionnelle** *('Tant que … faire …')* **:**

  4. Tant qu'il reste de l'huile, l'ajouter petit-à-petit en remuant

Spécification d'un algorithme

- Le nom, les paramètres d'entrée et la valeur de sortie s'appelle **la spécification d'un algorithme**
- Avec un éventuel commentaire supplémentaire, c'est tout ce qu'un utilisateur a à connaître.

Exemple :

<u>ALGOD&C</u>
- **Entrées :** x un réel, n un entier
- **Sortie :** un réel, correspondant à la valeur x^n
- **Commentaires :** *l'algo a une complexité en temps et en espace en* $O(\log n)$

Spécification d'un algorithme

- Le nom, les paramètres d'entrée et la valeur de sortie s'appelle **la spécification d'un algorithme**
- Avec un éventuel commentaire supplémentaire, c'est tout ce qu'un utilisateur a à connaître.

  Exemple :

ALGOD&C
- **Entrées :** x un réel, n un entier
- **Sortie :** un réel, correspondant à la valeur x^n
- **Commentaires :** *l'algo a une complexité en temps et en espace en* $O(\log n)$

- Dans une implémentation de l'algorithme, quelque soit le language, ce qui correspond à la spécification (nom, entrées, sortie) s'appelle **la signature** de la fonction correspondante.

1. Exemple introductif : calculer x^n

2. Un algorithme ?

3. Modèle pour la complexité algorithmique

4. Conception et analyse d'un algorithme

5. Outils mathématiques

Pourquoi un modèle ?

- **Dans ce cours, on ne va s'intéresser qu'à la complexité en temps.**

Pourquoi un modèle ?

- **Dans ce cours, on ne va s'intéresser qu'à la complexité en temps.**
- Un modèle pour répondre à la question :
  Quel temps va nécessiter la résolution d'un problème algorithmique ?

Pourquoi un modèle ?

- **Dans ce cours, on ne va s'intéresser qu'à la complexité en temps.**
- 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 ?

- **Dans ce cours, on ne va s'intéresser qu'à la complexité en temps.**
- 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 ?

- **Dans ce cours, on ne va s'intéresser qu'à la complexité en temps.**
- 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*.
- Deux modéles étudiés :

Modéle WORD-RAM (le cadre de ce cours) : 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 modéles étudiés :

Modéle WORD-RAM (le cadre de ce cours) : chaque **opération élémentaire** prend un **temps constant**

Modéle RAM : seulement chaque **opération sur un bit (ou un 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 modéles étudiés :

Modéle WORD-RAM (le cadre de ce cours) : chaque **opération élémentaire** prend un **temps constant**

Modéle RAM : seulement chaque **opération sur un bit (ou un chiffre)** prend un **temps constant**

- Exo : Pour chaque modèle, quel est le temps nécessaire pour lire le nombre n ?

Définition de la complexité

Sauf mention contraire, on se place dans le modèle WORD-RAM.
Dans ce modèle-là, on va :

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

Définition de la complexité

Sauf mention contraire, on se place dans le modèle WORD-RAM.
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

Sauf mention contraire, on se place dans le modèle WORD-RAM.
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

Sauf mention contraire, on se place dans le modèle WORD-RAM. 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.

Quelques remarques

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

- **Temps :** d'autres mesures existent (espace mémoire, temps parallèle, ...)
- **Pire cas :** raffinements possibles (en moyenne, analyses amortie et lissée, cas pratiques, ...)
- **Asymptotique :** On va 'cacher les constantes' (dans les O), mais en pratique, elles peuvent avoir leur importance...

Quelques remarques

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

- **Temps :** d'autres mesures existent (espace mémoire, temps parallèle, ...)
- **Pire cas :** raffinements possibles (en moyenne, analyses amortie et lissée, cas pratiques, ...)
- **Asymptotique :** On va 'cacher les constantes' (dans les O), mais en pratique, elles peuvent avoir leur importance...

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

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 ne va pas trop le faire 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 ne va pas trop le faire dans ce cours.
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 !).
   On ne va pas trop le faire dans ce cours.
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 !).
   On ne va pas trop le faire dans ce cours.
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 !).
   On ne va pas trop le faire dans ce cours.
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.
     Si $\mathcal{P}_i$ correspond à la diminution d'une valeur (*Ex. :* $\mathcal{P}_i =$ '$x \leq 2i - 1$'), on dit que cette valeur est un **variant**
   - Preuve par **récurrence**

Validité d'un algorithme

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

2. Exemple emprûnté à nos collègues lyonnais, préparateurs du DIU et popularisé par G. Berry...

Validité d'un algorithme

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

  Exemple :[2] dans l'appli *Zune* (pour lecteur MP3 Microsoft), il y avait une fonction donnant le numéro du jour de l'année à partir de la durée en jours depuis le 1er janvier 2004 (version simplifiée).

```python
def JourDeLAnnee(jour, annee=2004):
    while jour >365:
        if estBissextile(annee):
            if jour >366:
                jour=jour-366
                annee = annee+1
        else:
            jour=jour-365
            annee = annee+1
    return (jour,annee)
```

  Lancée en 2006, l'appli a planté le 31 décembre 2008.. ?..

  (appel : *jourDeLAnnee(1827)*, avec $1827 = 366 + 365 + 365 + 365 + 366$)

2. Exemple emprûnté à nos collègues lyonnais, préparateurs du DIU et popularisé par G. Berry...

Validité d'un algorithme

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

  Exemple :[2] dans l'appli *Zune* (pour lecteur MP3 Microsoft), il y avait une fonction donnant le numéro du jour de l'année à partir de la durée en jours depuis le 1er janvier 2004 (version simplifiée).

```python
def JourDeLAnnee(jour, annee=2004):
    while jour >365:
        if estBissextile(annee):
            if jour >366:
                jour=jour-366
                annee = annee+1
        else:
            jour=jour-365
            annee = annee+1
    return (jour,annee)
```

  Lancée en 2006, l'appli a planté le 31 décembre 2008.. ?..

  (appel : *jourDeLAnnee(1827)*, avec $1827 = 366 + 365 + 365 + 365 + 366$)

- D'autres exemples :

  `https://en.wikipedia.org/wiki/List_of_software_bugs`

2. Exemple emprûnté à nos collègues lyonnais, préparateurs du DIU et popularisé par G. Berry...

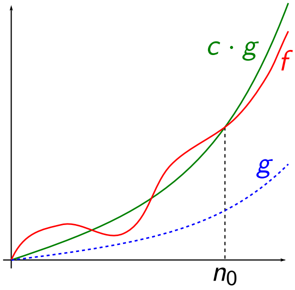
Notations de Landau

« Grand O »

Soit $f, g : \mathbb{N} \to \mathbb{R}_+$. Alors $\boldsymbol{f = O(g)}$ 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 $c \cdot g(n)$ »



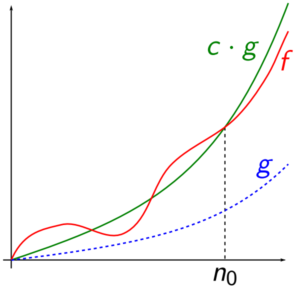
Notations de Landau

« Grand O »

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

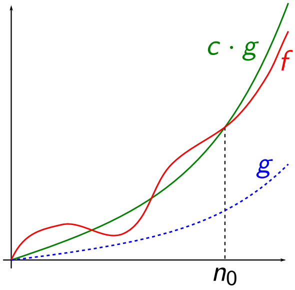
Notations de Landau

« Grand O »

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

Remarque : A priori, au programme de la spécialité NSI de terminale (avant, on ne parle que vaguement de 'coût'...)

Utilisation en complexité

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

$\rightsquigarrow$ si n est assez grand, le nb. d'opérations est $\leq$ constante$\times n^2$

Utilisation en complexité

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

$\rightsquigarrow$ si n est assez grand, le nb. d'opérations est $\leq$ 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) »

$\rightsquigarrow$ si n est assez grand, le nb. d'opérations est $\leq$ 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) »

$\rightsquigarrow$ si n est assez grand, le nb. d'opérations est $\leq$ 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 $n \geq 8$, $5n + 15 \leq n^2$ $\rightsquigarrow c = 1$ et $n_0 = 8$
- Ou alors pour $n \geq 3$, $5n + 15 \leq 5n^2$ $\rightsquigarrow c = 5$ et $n_0 = 3$

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 $n_0 = 3$

```
<inst. 1>;
pour i = 1 à n faire
    <inst. 2>;
pour i = 1 à n faire
    pour j = 1 à n
     faire
        <inst. 3>;
retourner var
```

- <inst. N> : opérations élémentaires

Exemples

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

- Car pour $n \geq 8$, $5n + 15 \leq n^2$ $\quad \leadsto c = 1$ et $n_0 = 8$
- Ou alors pour $n \geq 3$, $5n + 15 \leq 5n^2$ $\quad \leadsto c = 5$ et $n_0 = 3$

```
1 <inst. 1>;
2 pour i = 1 à n faire
3     <inst. 2>;
4 pour i = 1 à n faire
5     pour j = 1 à n faire
6         <inst. 3>;
7 retourner 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 $g : \mathbb{N} \to \mathbb{R}_+^*$ *(*$g(n) > 0$*). S'il existe une constante* $c \geq 0$ *telle que* $\lim_{n \to +\infty} \left(\frac{f(n)}{g(n)} \right) = c$*, alors* $f = O(g)$*.*

« Grand O » et limites

Lemme

Soit $g : \mathbb{N} \to \mathbb{R}_+^*$ *(*$g(n) > 0$*). S'il existe une constante* $c \geq 0$ *telle que* $\lim_{n\to+\infty} \left(\frac{f(n)}{g(n)}\right) = c$*, alors* $f = O(g)$*.*

Preuve

Si $f(n)/g(n) \longrightarrow_\infty c$, alors $f(n)/g(n) \leq c + 1$ à partir d'un certain rang. Donc $f(n) \leq (c+1)g(n)$, et $f = O(g)$.

« Grand O » et limites

Lemme

Soit $g : \mathbb{N} \to \mathbb{R}_+^*$ *(*$g(n) > 0$*). S'il existe une constante* $c \geq 0$ *telle que* $\lim_{n\to+\infty}\left(\frac{f(n)}{g(n)}\right) = c$*, alors* $f = O(g)$*.*

Preuve

Si $f(n)/g(n) \longrightarrow_\infty c$, alors $f(n)/g(n) \leq c+1$ à partir d'un certain rang. Donc $f(n) \leq (c+1)g(n)$, et $f = O(g)$.

Lemme

Soit $g : \mathbb{N} \to \mathbb{R}_+^*$ *(*$g(n) > 0$*). Si* $\lim_{n\to+\infty}\left(\frac{f(n)}{g(n)}\right) = +\infty$*, alors* $f \neq O(g)$*.*

« Grand O » et limites

Lemme

Soit $g : \mathbb{N} \to \mathbb{R}_+^*$ ($g(n) > 0$). S'il existe une constante $c \geq 0$ telle que $\lim_{n\to+\infty} \left(\frac{f(n)}{g(n)}\right) = c$, alors $f = O(g)$.

Preuve

Si $f(n)/g(n) \longrightarrow_\infty c$, alors $f(n)/g(n) \leq c + 1$ à partir d'un certain rang. Donc $f(n) \leq (c+1)g(n)$, et $f = O(g)$.

Lemme

Soit $g : \mathbb{N} \to \mathbb{R}_+^*$ ($g(n) > 0$). Si $\lim_{n\to+\infty} \left(\frac{f(n)}{g(n)}\right) = +\infty$, alors $f \neq O(g)$.

Preuve

Si $f(n)/g(n) \to_{+\infty} +\infty$, alors pour tout c, $f(n)/g(n) \geq c$ à partir d'un certain rang. Donc $f(n) \geq cg(n)$. Donc aucune constante c ne fonctionne, et $f \neq O(g)$.

« 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

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

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 tout $\alpha, \beta > 0$,

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

Si $\alpha < \beta$:

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

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 tout $\alpha, \beta > 0$,

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

Si $\alpha < \beta$:

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

Exemples

- $\log^2 n = O(\sqrt{n})$, $n^2 = O(n^3)$, ...
- Mais $\sqrt{n} \neq O(\log^2 n)$, $n^3 \neq O(n^2)$, ...
- $n^2(\log n)^4 = O(n^3)$
- $(\log n)^5 + n(\log n)^2 + n^3 \log n = O(n^3 \log n)$

Exemples de complexités de problèmes algorithmiques

Complexité	Notation 'O'	Exemple d'opération
Constante	$O(1)$	Initialisation d'une variable (dans le modèle WORD-RAM)
Logarithmique	$O(\log n)$	Recherche dichotomique dans un tableau trié
Linéaire	$O(n)$	Parcours d'un tableau (ou d'une liste)
N-log-N	$O(n \log n)$	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 $\{1, \ldots, n\}$
Factorielle	$O(n!)$	Énumération de toutes les permutations de $\{1, \ldots, n\}$

Exemples de complexités de problèmes algorithmiques

Complexité	Notation 'O'	Exemple d'opération
Constante	$O(1)$	Initialisation d'une variable (dans le modèle WORD-RAM)
Logarithmique	$O(\log n)$	Recherche dichotomique dans un tableau trié
Linéaire	$O(n)$	Parcours d'un tableau (ou d'une liste)
N-log-N	$O(n \log n)$	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 $\{1, \ldots, n\}$
Factorielle	$O(n!)$	Énumération de toutes les permutations de $\{1, \ldots, n\}$

Remarques :

- Parfois, on est content si on a un algorithme de résolution d'un problème de complexité polynomiale, on parle alors d'**algorithme polynomial**.
- Parfois, on a du mal à être content (voir Bloc 4, Algo Avancée).

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

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

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

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

Exemples

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

Autres outils mathématiques

Factorielle et formule de Stirling

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

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

Autres outils mathématiques

Factorielle et formule de Stirling

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

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

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

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

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

Sommes arithmétique et géométrique

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

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

Exemple :

$S \leftarrow 0$;
pour $i = 1$ *à* n **faire**
  $y \leftarrow 1$;
  pour $j = 1$ *à* $\boldsymbol{i}$ **faire**
    $y \leftarrow x \times y$;
  $S \leftarrow S + y$;
retourner S

Sommes arithmétique et géométrique

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

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

Exemple :

$S \leftarrow 0$;
pour $i = 1$ *à* n **faire**
  $y \leftarrow 1$;
  pour $j = 1$ *à* $\boldsymbol{i}$ **faire**
    $y \leftarrow x \times y$;
  $S \leftarrow S + y$;
retourner S

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

Sommes arithmétique et géométrique

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

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

Exemple :

$S \leftarrow 0$;
pour $i = 1$ *à* n **faire**
  $y \leftarrow 1$;
  pour $j = 1$ *à* $\boldsymbol{i}$ **faire**
    $y \leftarrow x \times y$;
  $S \leftarrow S + y$;
retourner S

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

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