Algorithmique avancée
Analyse des algorithmes

Youness LAGHOUAOUTA

Institut National des postes et télécommunications
laghouaouta@inpt.ac.ma

Fevrier 2022

Objectifs du cours

- Comprendre l'utilité des outils d'analyse des algorithmes
- Choisir les bonnes structures de données pour concevoir un algorithme
- Concevoir de nouvelles structures de données efficaces
- Comprendre certaines stratégies et les utiliser pour concevoir des algorithmes efficaces

Organisation du cours

1. Analyse des algorithmes + TD1
2. Algorithmes de tri + TD2
3. Structures de données linéaires + TD3
4. TD4
5. Arbres + TD5
6. Dictionnaires +TD6
7. Paradigmes et stratégies algorithmiques + TD7

Analyse des algorithmes

- L'exécution de l'algorithme produit-elle un résultat en temps fini quelles que soient les données fournies ? (**Terminaison**)
- Si l'algorithme termine en donnant une proposition de solution, alors cette solution est-elle correcte ? (**Correction**)
- L'algorithme se termine-t-il en un temps borné et en utilisant un espace mémoire raisonnable ? (**Complexité**)
- ...

Terminaison

- Un algorithme non récursif et qui ne contient pas une structure répétitive termine forcément (descente infinie)
- Pour prouver que l'algorithme termine, il faut trouver une quantité qui diminue strictement

Variant d'une boucle

Un variant de boucle est une expression :

- entière
- positive
- qui décroît strictement à chaque itération

Terminaison d'une boucle

Une boucle possédant un variant de boucle termine

Exemple

```
entrée    a,b : entiers
sortie    q, r : entiers
précondition   a > 0 et b > 0
r := a
q := 0
tant que  r >= b  faire
      r := r − b
      q := q + 1
fin tant que
retourne  q, r
```

r est un variant de la boucle

Correction

- Un algorithme est correct s'il est conforme à sa spécification
- La spécification d'un programme est la description sans ambiguïté de la tâche que doit effectuer un programme et des cas permis
- Outils pour prouver la correction d'un algorithme :
  - Algorithmes itératifs : triplets de Hoare, invariants de boucle
  - Algorithmes récursifs : preuves par induction

Triplet de Hoare

Un code est correct si le triplet (de Hoare) {P} S {Q} est vrai

- P est la précondition : conditions que doivent remplir les entrées valides de l'algorithme
- Q est la postcondition : conditions qui expriment que le résultat de l'algorithme est correct

Correction partielle

Si P est vérifiée et si le programme se termine on peut assurer Q

Correction totale

Si P est vérifiée le programme se termine et on peut assurer Q

Séquence

$$\{P\}\ S1;\ S2;\ \ldots;\ Sn;\ \{Q\}$$

On vérifie les triplets :

$$\begin{gathered}\{P\}\ S1\ \{P_1\}\\ \{P_1\}S2\{P_2\}\\ \ldots\\ \{P_{n-1}\}Sn\{Q\}\end{gathered}$$

Règles d'inférence

Structure conditionnelle

$$\{P\} \text{ si } C \text{ alors } S1 \text{ sinon } S2 \ \{Q\}$$

Il faut prouver que les deux triplets sont corrects

$$\{P \text{ et } C\} \ S1 \ \{Q\}$$
$$\{P \text{ et non } C\} \ S2 \ \{Q\}$$

Structure répétitive

$$\{P\} \text{ tant que C faire S } \{Q\}$$

Il faut trouver une assertion particulière I appelée invariant de boucle qui décrit l'état du programme pendant la boucle tel que

$$\{P\} \Rightarrow \{I\}$$
$$\{\text{I et C}\} \text{ S } \{\text{I}\}$$
$$\{\text{I et non C}\} \Rightarrow \{Q\}$$

Invariant

Un invariant est une propriété P telle que si P est vérifiée à une itération, alors elle l'est à l'itération suivante -> induction

Exemple

```
entrée      a, b : entiers
sortie      q, r : entiers
précondition  a > 0 et b > 0
r := a
q := 0
tant que  r >= b  faire
      r := r − b
      q := q + 1
fin tant que
retourne  q, r
```

a = b * q + r est l'invariant de la boucle

Preuve par induction

- Prouver que l'algorithme est correct pour n'importe quelle instance du problème
- Instances du problème classées par un ordre de grandeur (ex. taille du tableau)
- Démarche de vérification :
  - Vérifier que l'algorithme est correct pour le cas de base
  - Supposer que l'algorithme est correct pour une instance quelconque et en déduire qu'il est correct pour l'instance suivante
    - Pour un algorithme récursif, on démontre que l'appel courant est correct en supposant que les appels récursifs sont corrects
  - Prouver la terminaison
    - *Trivial* pour un algorithme récursif (les appels récursifs s'appliquent sur des sous-problèmes)

Exemple - 2^n

```
fonction     puissance2
entrée    n : entier
sortie    p : entier
précondition   n>=0
si   n = 0
        p := 1
sinon
        p := 2 * puissance2(n-1)
fin si
retourne   p
```

- Cas de base : pour n=0, *puissance2*(0) renvoie 1 (Correct)
- Pour n>=1, on suppose que l'appel *puissance2*(n) est correct (renvoie 2^n), donc l'appel *puissance2*(n+1) = 2 * *puissance2*(n) = $2 * 2^n = 2^{n+1}$ (Correct)

Complexité algorithmique

- Un algorithme est un ensemble d'instructions permettant de transformer un ensemble de données en un ensemble de résultats et ce, en un nombre fini étapes
- Pour atteindre cet objectif, un algorithme utilise deux ressources d'une machine : **le temps** et **l'espace mémoire**

Complexité temporelle et spatiale

- **Complexité temporelle** : temps d'exécution
- **Complexité spatiale** : l'espace mémoire utilisé pendant l'exécution

Mesure de la complexité

Pour comparer plusieurs algorithmes, deux méthodes peuvent être utilisées : **empirique** ou **mathématique**

Complexité pratique et théorique

- **Complexité pratique** : mesure précise de la complexité pour une machine donnée
- **Complexité théorique** : un ordre de grandeur de la complexité exprimé de manière indépendante des conditions d'exécution (machine, langage, compilateur/interpréteur. . .)

Calcul de la complexité temporelle théorique

- Effectuer des calculs sur l'algorithme en lui-même
- Décompter le nombre d'opérations élémentaires effectuées par l'algorithme :
  - addition, soustraction...
  - affectation
  - renvoi d'une valeur
  - ...
- Estimer le temps d'exécution de l'algorithme (suivant l'hypothèse que toutes les opérations élémentaires sont à égalité de coût)
- Le temps d'exécution d'un algorithme dépend de la quantité de données en entrée (T(n))
  - Exemple : pour la recherche d'une valeur dans un tableau, la complexité temporelle dépend de la longueur du tableau

Calcul de la complexité temporelle théorique

- Complexité dans le **meilleur des cas** $T(n) = min\{T(i_n), i_n \in D_n\}$
- Complexité dans le **pire des cas** $T(n) = max\{T(i_n), i_n \in D_n\}$
- Complexité en **moyenne** $T(n) = \sum_{i_n \in D_n} Pr(i_n)T(i_n)$

On s'intéresse le plus souvent à la complexité dans le **pire des cas**

- i_n est une instance de taille n
- D_n l'ensemble des instances de taille n
- $Pr(i_n)$ la probabilité de rencontrer i_n

Calcul de la complexité temporelle théorique

Séquence

$$I_1$$
$$I_2$$
$$.$$
$$I_k$$

$$T(n) = \sum_{i=1}^{k} T_{I\,i}(n)$$

- $T(n)$ représente le nombre total d'instructions
- $T_{I\,i}(n)$ représente le nombre d'instructions dans le bloc I_i

Structure conditionnelle

Si Condition
Alors I_1
Sinon I_2

$$T(n) = T_{condition}(n) + max(T_{I1}(n), T_{I2}(n))$$

- $T(n)$ représente le nombre total d'instructions
- $T_{condition}(n)$ représente le nombre d'instructions nécessaires pour tester la condition
- $T_{Ii}(n)$ représente le nombre d'instructions dans le bloc I_i

Calcul de la complexité temporelle théorique

Boucle Pour

Pour k allant de i à n

I_k

- I_k est le bloc d'instruction à exécuter en fonction de la valeur courante de k

$$T(n) = \sum_{k=i}^{n}(constante + T_{I\,k}(n))$$

Calcul de la complexité temporelle théorique

Boucle Tant que

Tant que Condition
Faire I
Fin Tant que

$$T(n) = a * (T_{Condition}(n) + T_I(n))$$

- a est le nombre de fois que le bloc I est exécuté

Exemple

- Rechercher un élément dans une liste et retourner sa position (0 si l'élément n'est pas trouvé)
- La complexité dans ce cas est fonction de la longueur de la liste (notée n)

```
fonction  rechercheSequentielle
entrée  x : entier , l : tableau d'entiers
sortie  r : entier
n := taille(n)
r := 0
pour i allant de 1 à n faire
    si l[i] = x alors
        r = i
        stop
    fin si
fin pour
retourne r
```

Exemple

```
fonction    rechercheSequentielle
entrée    x : entier , l : tableau d'entiers
sortie    r : entier
n := taille(n)  (2)
r := 0  (1)
pour  i  allant de  1  à  n  faire    (1)
      si  l[i] = x  alors  (2)
            r = i     (1)
            stop  (1)
      fin si
fin pour
retourne  r
```

```
fonction    rechercheSequentielle
entrée    x : entier , l : tableau d'entiers
sortie    r : entier
n := taille(n)  (2)
r := 0  (1)
pour  i  allant de  1  à  n  faire    (1)
      si  l[i] = x  alors  (2)
            r = i       (1)
            stop   (1)
      fin si
fin pour
retourne  r
```

- Complexité dans le meilleur des cas : $T(n) = 8$
- Complexité dans le pire des cas : $T(n) = 3 + 3n$
- Complexité moyenne : $T(n) = 3 + \sum_{i=1}^{n} \frac{1}{n} 3i = 3 + \frac{3}{2}(n+1)$

Complexité asymptotique

- Le décompte d'instructions peut s'avérer fastidieux à effectuer si on tient compte d'autres instructions (E/S, opérateurs logiques, appels de fonctions. . .)
- Que la complexité d'un algorithme soit égale à 3n+2 ou 2n+6 n'affecte pas son efficacité

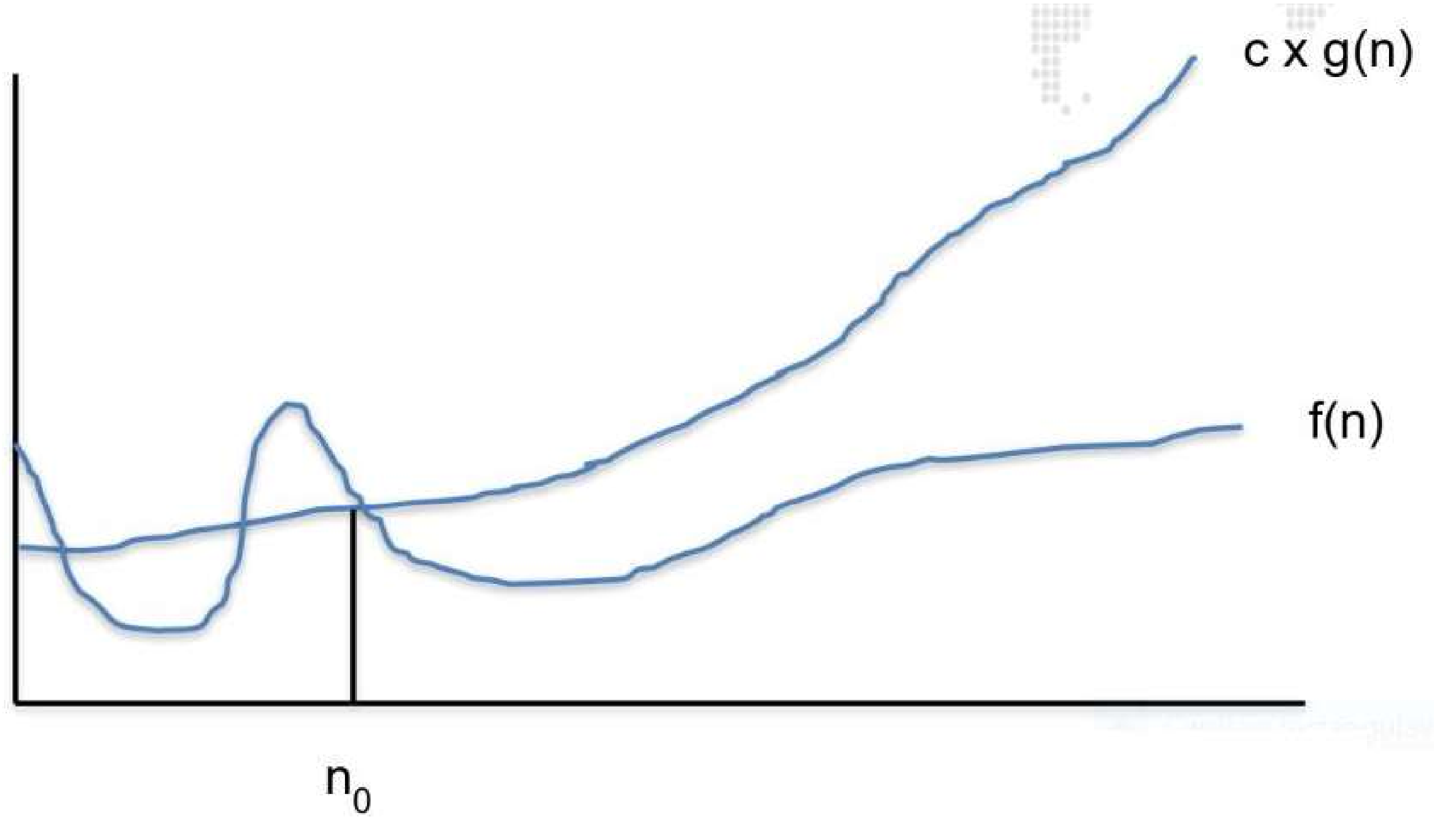
Complexité asymptotique

- S'intéresse à une approximation du nombre d'opérations élémentaires en ignorant toute constante pouvant apparaitre lors du décompte
- Permet d'avoir un résultat indicatif de la complexité tout en faisant des calculs gérables
- Décrit le comportement de l'algorithme quand la taille des données devient de plus en plus grande (vitesse de croissance)

Notation asymptotique

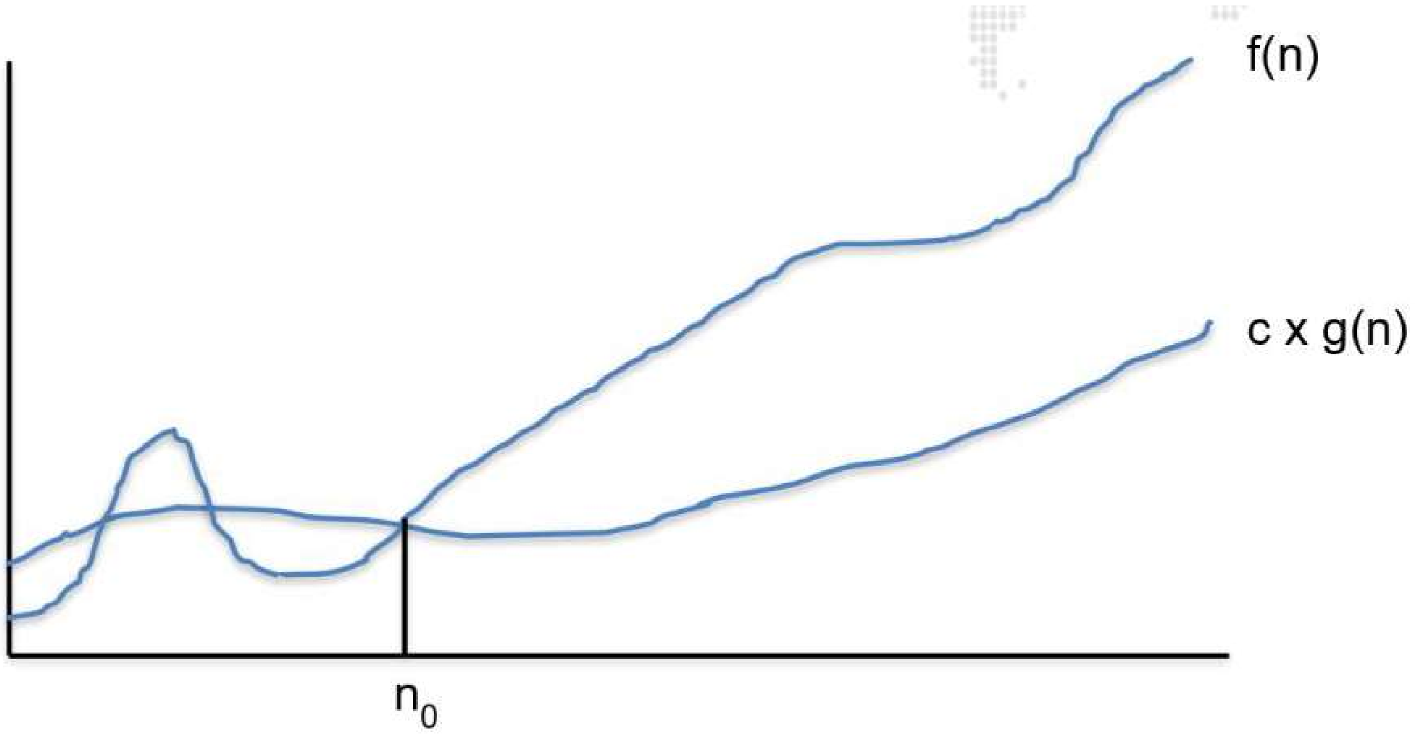
On dit qu'une fonction f est un **grand O** ($f(n) = O(g(n))$) d'une fonction g si est seulement si :

$$\exists c > 0, \exists n_0 > 0 / \forall n > n_0, f(n) < c * g(n) \tag{1}$$



On dit qu'une fonction f est un **grand Omega** ($f(n) = \Omega(g(n))$) d'une fonction g si est seulement si :

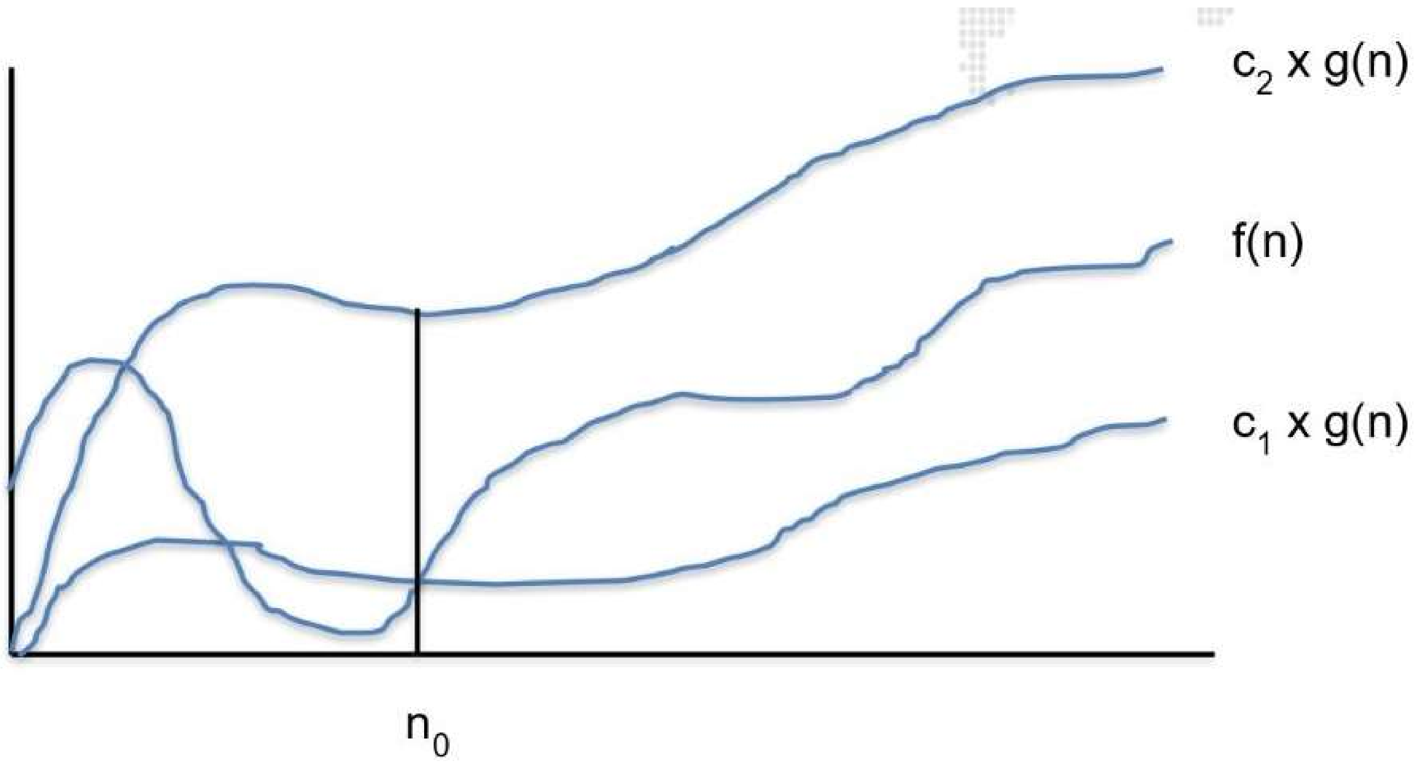
$$\exists c > 0, \exists n_0 > 0 / \forall n > n_0, f(n) > c * g(n) \quad (2)$$



Notation asymptotique

On dit qu'une fonction f est un **grand Theta** ($f(n) = \Theta(g(n))$) d'une fonction g si est seulement si :

$$\exists c_1 > 0, \exists c_2 > 0, \exists n_0 > 0 / \forall n > n_0, c_1 * g(n) < f(n) < c_2 * g(n) \quad (3)$$



Complexité asymptotique - Règles de simplification

- $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- $f(n) = O(k * g(n)) \Rightarrow f(n) = O(g(n))$
- $f1(n) = O(g1(n)) \wedge f2(n) = O(g2(n)) \Rightarrow O(f1(n) + f2(n)) = O(max(g1(n), g2(n)))$

Notation asymptotique - Exemples

- $T(n) = 4$ -> $O(1)$
- $T(n) = 3n + 2$ -> $O(n)$
- $T(n) = 4n^2 + 2n + 6$ -> $O(n^2)$
- $T(n) = 2log(n) + 4$ -> $O(log(n))$
- $T(n) = 3log(n) + 3n$ -> $O(n)$
- $T(n) = 2^n + 6n^3 + 4$ -> $O(2^n)$

Classes de complexité

- La complexité d'un algorithme s'exprime comme un grand O d'une fonction de référence
- Permet de définir des classes de complexité
- Des algorithmes appartement à la même classe sont de complexité équivalente

O	Type de complexité
$O(1)$	Constant
$O(log(n))$	Logarithmique
$O(n)$	Linéaire
$O(nlog(n))$	Quasi-linéaire
$O(n^2)$	Quadratique
$O(n^3)$	Cubique
$O(2^n)$	Exponentielle
$O(n!)$	Factorielle

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^{a>1}) \subset O(2^n) \subset O(n!)$$