Algorithmique procédurale

La complexité

Équipe pédagogique

CY Tech

Bibliographie - Sitographie

- Houcine Senoussi
- Eric Sopena (Université de Bordeaux)
- Olivier Baudon (Université de Bordeaux)
- Laurence Pilard (Université de Versailles)

- Voir aussi Rachid Guerraoui :
  `https://www.youtube.com/watch?v=clZ4q5zPBlE`

Rappels

Définition d'un algorithme

Un algorithme est une procédure de calcul définie prenant en entrée une valeur ou un ensemble de valeurs et donnant en sortie une valeur ou un ensemble de valeurs.

- Un algorithme est donc une séquence d'étapes de calcul qui transforment l'entrée en sortie.

Exemples

- l'algorithme d'Euclide qui permet de calculer le p.g.c.d. de 2 entiers ;
- les algorithmes de tri permettant de ranger par ordre alphabétique ;
- les algorithmes de recherche d'une chaine de caractères dans un texte ;
- les algorithmes d'ordonnancement, qui permettent de d'écrire la coordination entre différentes taches nécessaires pour mener à bien un projet.

Rappels

- Un programme destiné à être exécuté par un ordinateur est, la plupart du temps, la description d'un algorithme dans un langage accepté par cette machine.

Rappels

- Un algorithme décrit un traitement sur un certain nombre, fini, de données.
- Un algorithme est la composition d'un ensemble fini d'étapes, chaque étape étant formée d'un nombre fini d'opérations dont chacune est :
  - définie de façon rigoureuse et non ambiguë ;
  - effective i.e. pouvant être effectivement réalisée par une machine.
- Un algorithme doit toujours se terminer après un nombre fini d'opérations, et fournir un résultat.

Objectifs

- L'objectif est d'apprendre à analyser ces algorithmes.
- Analyser un algorithme en trois étapes :
  1. **Terminaison** : prouver qu'il termine.
  2. **Correction** : prouver qu'il résout le problème.
  3. **Complexité** : évaluer son coût en temps et en espace.

Complexité des algorithmes

- On étudie donc des problèmes pour lesquels il existe des algorithmes et **on va analyser la complexité de ces algorithmes**, i.e. les ressources nécessaires à leur exécution, en temps et en mémoire.
- On ne s'intéresse qu'aux algorithmes de complexité "raisonnable".
- *Attention* : il ne suffit pas de savoir qu'il existe un algorithme pour être certain qu'il est utilisable pratiquement.

Complexité des algorithmes

Exemple du jeu d'échec

- En théorie, à chaque coup joué, il y a un nombre fini de coups possibles. On peut donc concevoir un programme qui calculerait toutes les conséquences de tous les coups possibles.
- Mais la complexité du problème est telle qu'il est difficile de mettre un tel algorithme en pratique.
- Il faudrait considérer de l'ordre de 10^{19} positions possibles pour décider un coup.
- Rappelons que 10^{19} millisecondes est de l'ordre de 300 millions d'années !

Complexité des algorithmes

- Etant donné un algorithme, toute exécution de cet algorithme sur les mêmes données donne lieu à la même suite d'opérations.
- Un algorithme peut être exprimé en langage courant ou pseudo-code (le français, par exemple).
- Un algorithme doit être exprimé dans un langage de programmation pour être compris et exécuté par un ordinateur.
- Un algorithme est indépendant du langage de programmation utilisé : l'algorithme d'Euclide programmé en C, en C++, en Java ou en tout autre langage de programmation reste toujours l'algorithme d'Euclide.

Complexité des algorithmes

- L'exécution d'un programme nécessite l'utilisation des ressources de l'ordinateur :
  - temps de calcul pour exécuter les opérations,
  - occupation de la mémoire pour contenir et manipuler le programme et ses données.
- La complexité d'un algorithme permet de quantifier deux grandeurs physiques en terme de **temps d'exécution** et de **place mémoire**, dans le but
  - d'évaluer l'efficacité d'un algorithme,
  - de comparer entre eux différents algorithmes qui résolvent le même problème.

Complexité des algorithmes

- Un algorithme performant est un compromis entre
  - le temps d'exécution,
  - l'espace mémoire utilisé.
- On doit déterminer quelle mesure utiliser pour calculer ces deux quantités.

Complexité des algorithmes

- Pour un programme sur une machine, on pourrait par exemple exprimer la complexité en temps (resp. en espace) par le nombre de cycles machines (resp. le nombre de bits) utilisés lors de l'exécution du programme en comptant :
  - pour le temps : le nombre d'opérations effectuées par le programme et le temps nécessaire pour chaque opération,
  - pour l'espace : le nombre d'instructions et le nombre de données du programme, avec le nombre de bits nécessaires pour stocker chacune d'entre elles, ainsi que le nombre de bits supplémentaires pour la manipulation des données.

Complexité des algorithmes

- Ce type d'analyse conduit à des énoncés comme :
  L'algorithme A implémenté par le programme P sur l'ordinateur O et exécuté sur la donnée D utilise
  - k secondes de calcul,
  - j bits de mémoire.
- Un résultat de ce genre peut être une source d'information intéressante.
- Mais ...

Complexité des algorithmes

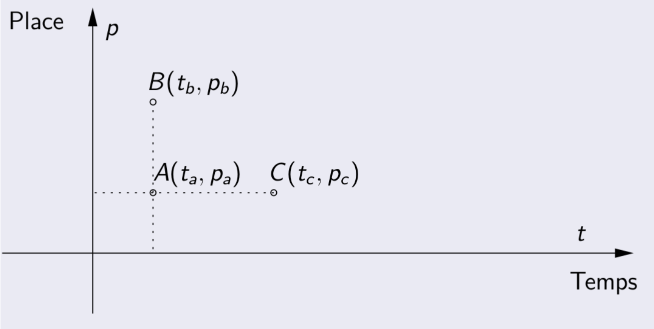
- Mais ...
- le but de l'analyse de la complexité des algorithmes est d'établir des résultats plus généraux permettant d'estimer l'efficacité de la méthode utilisée par un algorithme, indépendamment
  - de la machine,
  - du langage de programmation,
  - du compilateur,
  - de tous les détails d'implémentation.

Complexité des algorithmes

- Le type d'énoncé que l'on souhaite produire est :
  - Sur toute machine, et quel que soit le langage de programmation, l'algorithme A est meilleur que l'algorithme B pour les données de grande taille.

  ou encore,
  - L'algorithme A est optimal en nombre de comparaisons pour résoudre le problème P.

Complexité des algorithmes



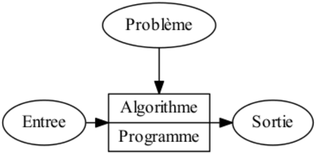
- Ici, l'algorithme A utilise un temps t_a et un espace p_a , il est donc plus efficace en temps que l'algorithme C et plus efficace en mémoire que l'algorithme B

Complexité des algorithmes

Soit P : un problème
Soit M : une méthode pour résoudre le problème P

- Un algorithme est une description de la méthode M dans un langage algorithmique.
- La théorie de la complexité (algorithmique) vise à répondre aux besoins suivants :
  - classer les problèmes selon leur difficulté,
  - classer les algorithmes selon leur efficacité,
  - comparer les algorithmes sans devoir les implémenter.

Complexité des algorithmes



- Souvent, il existe plusieurs algorithmes pour répondre à un problème donné. Chaque algorithme possède :
  - une complexité en temps : son temps d'exécution est proportionnel à la quantité de données fournies en entrée.
  - une complexité en espace : la quantité de mémoire utilisée par l'algorithme.
- Le bon algorithme (le plus efficace) est celui qui possède la complexité la plus faible. Il faut donc la mesurer.

Complexité des algorithmes

- La complexité temporelle (resp. spatiale) caractérise le coût en temps (resp. en espace) d'un algorithme.

Complexité en mémoire

- La complexité en mémoire d'un algorithme est la fonction qui associe à la taille du problème la taille moyenne de l'espace de mémoire nécessaire à sa résolution.

Complexité temporelle

- La complexité temporelle d'un algorithme est le nombre d'opérations élémentaires qui le composent.
  - On appelle opération élémentaire une opération dont le coût est constant (ne dépend pas du problème).

Complexité des algorithmes

Exemples d'opérations élémentaires

- affectation, addition, multiplication, comparaison, permutation.
- pour la recherche d'un élément dans une liste :
  - le nombre de comparaisons entre cet élément et les entrées de la liste.
- pour trier une liste d'éléments :
  - le nombre de comparaisons entre 2 éléments ;
  - le nombre de déplacements d'éléments.
- pour multiplier 2 matrices :
  - le nombre de multiplications et le nombre d'additions.

Complexité des algorithmes

- Ce nombre d'opérations est calculé en fonction de la taille du problème notée n.
- Taille d'un problème : taille d'une instance du problème, c'est-à-dire la taille des données que l'algorithme reçoit en entrée.
  - Exemple : dans le cas du tri la taille du problème = nombre d'éléments du tableau à trier.
- On donne souvent la complexité sous la forme $O(g(n))$
- Signification de cette notation : Le nombre d'opérations $f(n)$ a comme terme prépondérant (de plus haut degré) $g(n)$.
  - Exemples : $O(n^2)$, $O(n \log n)$, $O(2^n)$.

Complexité des algorithmes

- Après avoir déterminé les opérations élémentaires, il s'agit de compter le nombre d'opérations de chaque type.
  - Chaque instruction basique (affectation d'une variable, comparaison, ...) va consommer une unité de temps ;
  - Chaque itération d'une boucle rajoute le nombre d'unités de temps consommées dans le corps de cette boucle ;
  - Chaque appel de fonction rajoute le nombre d'unités de temps consommées dans cette fonction ;
- Pour avoir le nombre d'opérations effectuées par l'algorithme, on additionne le tout.

Complexité des algorithmes

- Évaluation du nombre d'opérations élémentaires en fonction de :
  - la taille des données,
  - de la nature des données.
- Notations :
  - n : taille des données,
  - $T(n)$: nombre d'opérations élémentaires
- Configurations possibles :
  - meilleur cas, pire des cas et cas moyen

$T(n)$: **nombre d'opérations élémentaires**

- Instruction élémentaire
  - L'exécution d'une instruction élémentaire consomme une unité de temps ;
  - L'exécution d'une opération de mémorisation consomme une unité d'espace.

Complexité des algorithmes

$T(n)$: **nombre d'opérations élémentaires**

- Séquence d'instructions
  - Lorsque les opérations élémentaires sont dans une séquence d'instructions, leurs nombres s'ajoutent.
- Branchement conditionnel
  - Pour les branchement conditionnels, on peut majorer le nombre d'opérations élémentaires.

Évaluation de $T(n)$ (séquence)

- On fait la somme des coûts

$$\left.\begin{array}{ll} \text{Traitement1} & T_1(n) \\ & \\ \text{Traitement2} & T_2(n) \end{array}\right\} \quad T(n) = T_1(n) + T_2(n)$$

Évaluation de $T(n)$ (embranchement)

- On prend le maximum des coûts

$$\left.\begin{array}{ll} \textbf{si} \ < \text{condition} > \ \textbf{alors} & \\ \qquad \text{Traitement1} & T_1(n) \\ \textbf{sinon} & \\ \qquad \text{Traitement2} & T_2(n) \end{array}\right\} max(T_1(n), T_2(n))$$

Évaluation de $T(n)$ (boucle)

- Pour les boucles, le nombre d'opérations élémentaires dans la boucle est la somme du nombre d'opérations élémentaires $T_i(n)$ pour chaque itérations.
- On prend la somme des coûts des itérations successives

$$\left.\begin{array}{ll} \textbf{tant que} \; < \text{condition} > \; \textbf{faire} & \\ \quad\quad \text{Traitement} \quad\quad T_i(n) & \\ \textbf{fin faire} & \end{array}\right\} \quad \sum_{i=1}^{k} T_i(n)$$

$T_i(n)$: coût de la $i^{\text{ème}}$ itération

Procédure, fonction

- Pour les appels de procédures ou de fonction :
  - s'il n'y a pas de procédures ou de fonctions récursives : c'est leur nombre d'opérations fondamentales.
  - pour des fonctions récursives : compter le nombre d'opérations fondamentales donne en général lieu à la résolution de relations de récurrence.
    En effet $T(n)$ s'écrit, selon la récursion, en fonction de $T(k)$ pour $k < n$.

Exemple

- Soit le problème :
- Étant donné $n \in \mathbb{N}^*$ donné au clavier par l'utilisateur, calculer n^n

Complexité des algorithmes

Exemple

- Soit le problème :
- Étant donné $n \in \mathbb{N}^*$ donné au clavier par l'utilisateur, calculer n^n

Algorithme de calcul

lire(n)
si n $\leq$ 1 **alors**
  écrire("impossible")
sinon
  x $\leftarrow$ n
  y $\leftarrow$ n-1
  répéter
    x $\leftarrow$ x*n
    y $\leftarrow$ y-1
    écrire(x)
  jusqu'à y=0
fin si

Complexité des algorithmes

Exemple

- Opérations élémentaires :
- Lecture :

Algorithme de calcul

lire(n)
si n $\leq$ 1 **alors**
    écrire("impossible")
sinon
    x $\leftarrow$ n
    y $\leftarrow$ n-1
    répéter
        x $\leftarrow$ x*n
        y $\leftarrow$ y-1
        écrire(x)
    jusqu'à y=0
fin si

Complexité des algorithmes

Exemple

- Opérations élémentaires :
- Lecture : 1

Algorithme de calcul

lire(n)
si n $\leq$ 1 **alors**
  écrire("impossible")
sinon
  x $\leftarrow$ n
  y $\leftarrow$ n-1
  répéter
    x $\leftarrow$ x*n
    y $\leftarrow$ y-1
    écrire(x)
  jusqu'à y=0
fin si

Complexité des algorithmes

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel :

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Complexité des algorithmes

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
    écrire("impossible")
sinon
    x ← n
    y ← n-1
    répéter
        x ← x*n
        y ← y-1
        écrire(x)
    jusqu'à y=0
fin si
```

Complexité des algorithmes

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture :

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Complexité des algorithmes

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n
- Affectation :

Algorithme de calcul

lire(n)
si $n \leq 1$ **alors**
    écrire("impossible")
sinon
    $x \leftarrow n$
    $y \leftarrow n-1$
    répéter
        $x \leftarrow x*n$
        $y \leftarrow y-1$
        écrire(x)
    jusqu'à y=0
fin si

Complexité des algorithmes

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n
- Affectation : 2

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n
- Affectation : 2+ 2.(n-1) = 2n

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Complexité des algorithmes

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n
- Affectation : 2+ 2.(n-1) = 2n
- Soustraction :

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n
- Affectation : 2+ 2.(n-1) = 2n
- Soustraction : 1

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Complexité des algorithmes

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n
- Affectation : 2+ 2.(n-1) = 2n
- Soustraction : 1+ (n-1) = n

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n
- Affectation : 2+ 2.(n-1) = 2n
- Soustraction : 1+ (n-1) = n
- Multiplication :

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n
- Affectation : 2+ 2.(n-1) = 2n
- Soustraction : 1+ (n-1) = n
- Multiplication : n-1

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n
- Affectation : 2+ 2.(n-1) = 2n
- Soustraction : 1+ (n-1) = n
- Multiplication : n-1
- Boucle :

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n
- Affectation : 2+ 2.(n-1) = 2n
- Soustraction : 1+ (n-1) = n
- Multiplication : n-1
- Boucle : n-1

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Complexité des algorithmes

Exemple

- Opérations élémentaires :
- Lecture : 1
- Branchement conditionnel : 1
- Écriture : 1+ (n-1) = n
- Affectation : 2+ 2.(n-1) = 2n
- Soustraction : 1+ (n-1) = n
- Multiplication : n-1
- Boucle : n-1
- On a donc une complexité en temps de : 1 + 1 + n + 2n + n + n-1 + n-1 = 6n

Algorithme de calcul

```
lire(n)
si n ≤ 1 alors
   écrire("impossible")
sinon
   x ← n
   y ← n-1
   répéter
      x ← x*n
      y ← y-1
      écrire(x)
   jusqu'à y=0
fin si
```

Autre exemple

- Quelle est la complexité en temps de l'algorithme suivant qui permet de rechercher un élément X dans une séquence A composée de n éléments

```
j ← 1
tant que (j ≤ n) et (A[j] ≠ X) faire
    j ← j+1
fin tant que
si j>n alors
    j ← -1
fin si
```

Autre exemple

j $\leftarrow$ 1
tant que (j $\leq$ n) **et** (A[j] $\neq$ X) **faire**
  j $\leftarrow$ j+1
fin tant que
si j>n **alors**
  j $\leftarrow$ -1
fin si

- On peut tout calculer mais aussi ne prendre que les éléments significatifs comme :
  - le nombre d'itération (nombre de tours dans la boucle tant que) et
  - le nombre d'opérations par itérations.

Autre exemple

j $\leftarrow$ 1
tant que (j $\leq$ n) **et** (A[j] $\neq$ X) **faire**
  j $\leftarrow$ j+1
fin tant que
si j>n **alors**
  j $\leftarrow$ -1
fin si

- Mais le nombre d'itération n'est pas constant car :
  - si $X \in A$, alors ce nombre est égal à n
  - sinon il est égal à j, le rang de la première occurrence de X

Autre exemple

j ← 1
tant que (j ≤ n) **et** (A[j] ≠ X) **faire**
  j ← j+1
fin tant que
si j>n **alors**
  j ← -1
fin si

- Donc il y a au plus n itérations mais il n'y en a que $j \leq n$ si $X \in A$
- On obtient : $1 \leq T(n) \leq n$

Complexité des algorithmes

- L'exemple précédent met bien en évidence, que le temps d'exécution d'un algorithme dépend de la donnée sur laquelle il opère.

Complexité des algorithmes

Influence de la taille des données

Problèmes :

- addition et multiplication
- recherche ou tri
- produit de matrices carrées
- parcours d'arbres ou de graphes

Taille :

- le nombre de chiffres des nombres
- le nombre d'éléments manipulés
- la dimension de la matrice
- le nombre de sommets ou le nombres d'arcs (d'arêtes)

Complexité au mieux et au pire

- Comportement d'un algorithme sur l'ensemble D_n des données de taille n
- Notons $t_A(d)$ la complexité en temps de l'algorithme A sur la donnée $d \in D_n$
- La complexité dans le meilleur des cas :

$$Min_A(n) = min\{t_A(d) \mid d \in D_n\}$$

- La complexité dans le pire des cas :

$$Max_A(n) = max\{t_A(d) \mid d \in D_n\}$$

Complexité des algorithmes

- Les complexités dans le **meilleur** et dans le **pire** des cas donnent des indications sur les **bornes extrêmes** de la complexité de l'algorithme sur les données de taille n.
- La complexité dans le **pire des cas**, qui donne une **borne supérieure** du temps d'exécution, est particulièrement utile car elle permet de donner une estimation de la taille maximale des données qui pourront être traitées par l'algorithme.

Retour sur l'exemple

j $\leftarrow$ 1
tant que (j $\leq$ n) **et** (A[j] $\neq$ X) **faire**
  j $\leftarrow$ j+1
fin tant que
si j>n **alors**
  j $\leftarrow$ -1
fin si

- $Max_A(n) = n$ (pire des cas)
- $Min_A(n) = j$ (meilleur des cas)

Complexité des algorithmes

- Les cas extrêmes ne sont pas les plus fréquents et dans la pratique on aimerait savoir quel comportement attendre en général de l'algorithme, d'où l'introduction de la **complexité en moyenne**

Complexité des algorithmes

Comportement d'un algorithme sur l'ensemble D_n des données de taille n

- Soit D_n l'ensemble de toutes les données possibles de taille n et notons $t_A(d)$ la complexité en temps de l'algorithme A sur la donnée $d \in D_n$.
- La complexité en moyenne :

$$Moy_A(n) = \sum_{d \in D_n} p(d) t_A(d)$$

où $p(d)$ est la probabilité que l'on ait la donnée d en entrée de l'algorithme.

Comportement d'un algorithme sur l'ensemble D_n des données de taille n

- Nécessite de connaître (ou de pouvoir estimer) la distribution des données
- Souvent difficile à calculer
- Intéressant si le comportement "usuel" de l'algorithme est éloigné du pire cas

$$Min_A(n) \leq Moy_A(n) \leq Max_A(n)$$

Complexité des algorithmes

- La complexité dans le pire des cas correspond à une borne supérieure du temps d'exécution pour une entrée quelconque.
- D'où la certitude que le programme ne mettra jamais plus de temps que cette valeur.
- Pour beaucoup d'algorithmes le cas le plus défavorable survient très souvent.
- Souvent, le cas moyen est presque aussi mauvais que le cas le plus défavorable.

Complexité des algorithmes

- On a déterminé la complexité d'un algorithme comme une fonction de la taille des données $T(n)$.
- Il est très important de connaître la rapidité de croissance de cette fonction lorsque la taille des données croît.
- Pour traiter un problème de petite taille la méthode employée importe peu, alors que **pour un problème de grande taille**, les différences de performance entre algorithmes peuvent être énormes.

- Souvent, une simple approximation de la fonction de complexité suffit pour savoir si un algorithme est utilisable ou non, ou pour comparer entre eux différents algorithmes.
- **Pour n grand, il est souvent secondaire de savoir si un algorithme fait $n + 1$ ou $n + 2$ opérations**

- Les constantes multiplicatives ont peu d'importance
- Supposons que l'on ait à comparer :
  - l'algorithme A_1 de complexité $T_1(n) = k_1 n^2$
  - l'algorithme A_2 de complexité $T_2(n) = k_2 n$

  Quelles que soient les constantes multiplicatives k_1 et k_2, l'algorithme A_2 est toujours meilleur que A_1 à partir d'un certain n, car la fonction $f(n) = n^2$ croît beaucoup plus vite que la fonction $g(n) = n$.

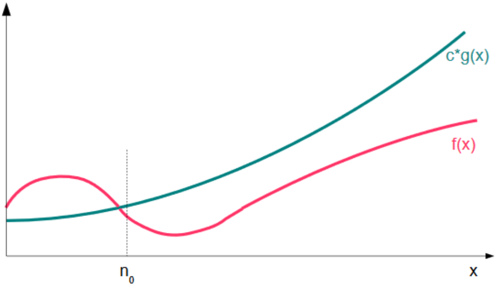
- On dit que l'**ordre de grandeur asymptotique** de $f(n)$ est strictement plus grand que celui de $g(n)$
- En effet : $\lim\limits_{n \to \infty} \dfrac{g(n)}{f(n)} = 0$

- Pour comparer les ordres de grandeur asymptotiques des fonctions, on utilise la définition suivante (**notation de Landau**) :
- Étant donné des fonctions $f(n)$ et $g(n)$, on dit que $f(n)$ est $O(g(n))$ s'il existe des constantes $c > 0$ et $n_0 \geq 1$ telles que $\forall n \in \mathbb{N}, n \geq n_0, f(n) \leq c.g(n)$
- $f \in O(g)$ se prononce "f est en grand O de g"

Complexité des algorithmes

Ordre de grandeur asymptotique

f et g étant des fonctions, f = O(g) s'il existe des constantes c>0 et n_0 telles que f(x) < c*g(x) pour tout x > n_0



f = O(g) signifie que f est dominée asymptotiquement par g ou que g domine asymptotiquement f.

- La notation grand O donne une borne supérieure du taux de croissance d'une fonction.
- "$f(n)$ est $O(g(n))$" signifie donc que le taux de croissance de $f(n)$ est plus petit ou égal au taux de croissance de $g(n)$.
- On peut utiliser la notation O pour ordonner les fonctions à partir de leur taux de croissance.

La notation grand O vérifie les propriétés suivantes :

- si $f = O(g)$ et $g = O(h)$ alors $f = O(h)$
- si $f = O(g)$ et k un nombre, alors $k * f = O(g)$
- si $f_1 = O(g_1)$ et $f_2 = O(g_2)$ alors $f_1 + f_2 = O(g_1 + g_2)$
- si $f_1 = O(g_1)$ et $f_2 = O(g_2)$ alors $f_1 * f_2 = O(g_1 * g_2)$

Exemples de domination asymptotique

- $x = O(x^2)$ car pour $x > 1, \quad x < x^2$
- $x^2 = O(x^3)$ car pour $x > 1, \quad x^2 < x^3$
- $100 * x = O(x^2)$ car pour $x > 100, \quad x < x^2$
- $\ln(x) = O(x)$ car pour $x > 0, \quad \ln(x) < x$
- si $i > 0, x^i = O(e^x)$ car pour x tel que $\dfrac{x}{\ln(x)} > i, \quad x^i < e^x$

Complexité des algorithmes

Principales classes de complexité

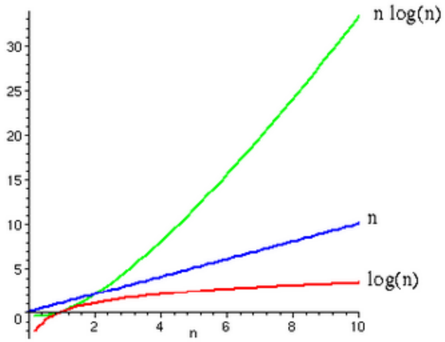
- $O(1)$: temps constant, pas d'augmentation du temps d'exécution quand le paramètre croit
- $O(\log(n))$: logarithmique, augmentation très faible du temps d'exécution quand le paramètre croit.
  - *Exemple* : algorithmes qui décomposent un problème en un ensemble de problèmes plus petits (dichotomie).
- $O(n)$: linéaire, augmentation linéaire du temps d'exécution quand le paramètre croit (si le paramètre double, le temps double).
  - *Exemple* : algorithmes qui parcourent séquentiellement des structures linéaires.

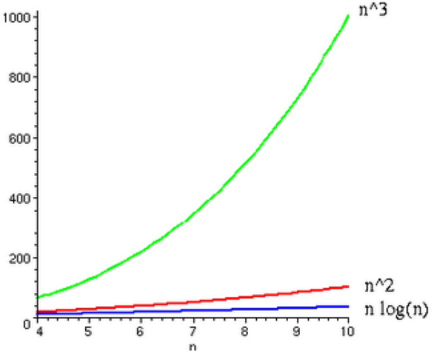
- $O(n\log(n))$: quasi-linéaire, augmentation un peu supérieure à $O(n)$.
  - *Exemple* : algorithmes qui décomposent un problème en d'autres plus simples, traités indépendamment et qui combinent les solutions partielles pour calculer la solution générale. Tris fusion et rapide.
- $O(n^2)$: quadratique, polynomial, quand le paramètre double, le temps d'exécution est multiplié par 4.
  - *Exemple* : algorithmes avec deux boucles imbriquées. Tris à bulle, par insertion et par sélection.
- $O(n^3)$: cubique, polynomial

- $O(n^i)$: polynomial avec i entier et $i \geq 2$, quand le paramètre n double, le temps d'exécution est multiplié par 2^i
  - *Exemple* : algorithme utilisant i boucles imbriquées
- $O(i^n)$: exponentielle avec $i > 1$ (problèmes très difficiles), quand le paramètre n double, le temps d'exécution est élevé à la puissance 2.
- $O(n!)$: factorielle
- $O(n^n)$

Complexité des algorithmes

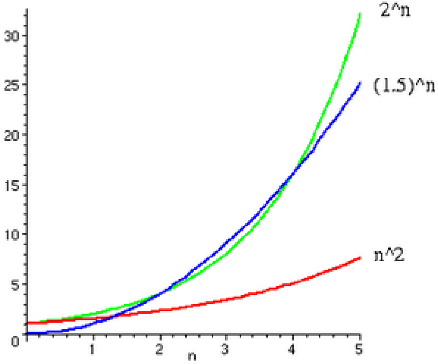
Ordre de grandeur asymptotique

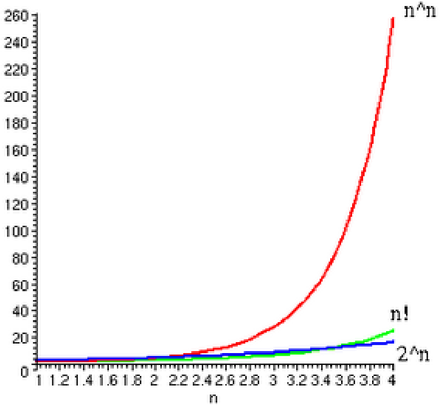




Complexité des algorithmes

Ordre de grandeur asymptotique





- Si on compare :
  - l'algorithme A_1 de complexité $TA_1(n)$
  - l'algorithme A_2 de complexité $TA_2(n)$
- Si l'ordre de grandeur de $TA_1(n)$ est strictement plus petit que l'ordre de grandeur de $TA_2(n)$, alors on peut conclure immédiatement que A_1 est meilleur que A_2 pour n grand.
- Par contre, si 2 fonctions ont le même ordre de grandeur asymptotique, il faut faire une analyse plus fine pour pouvoir comparer les 2 algorithmes.

Complexité des algorithmes

Exemples

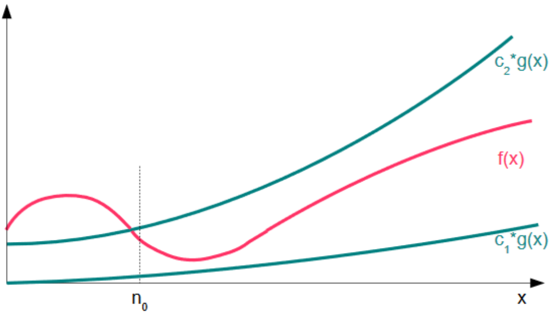
- $T(n) = n^3 + 2n^2 + 4n + 2 = O(n^3)$
  - Si $n = 2$, on a $T(n) = 26$ et $n^3 = 8$
  - Mais si $n = 10^6$, on a $T(n) = 10^{216}$ et $n^3 = 10^{216}$
- $T(n) = n\log(n) + 12n + 2 = O(n\log(n))$
- $T(n) = 2n^{10} + n^7 + 12n^4 + \dfrac{2^n}{100} = O(2^n)$

Définitions

- Un algorithme en temps polynomial est un algorithme dont le temps d'exécution est en $O(n^k)$.
- S'il existe des constantes c_1, c_2, strictement positives et n_0 telles que $c_1 * g(x) \leq f(x) \leq c_2 * g(x)$ pour tout $x \geq n_0$, on notera $f = \Theta(g)$
- S'il existe des constantes $c > 0$ et n_0 telles que $f(x) \geq c * g(x)$ pour tout $x \geq n_0$, on notera $f = \Omega(g)$

Complexité des algorithmes

f et g étant des fonctions, f = Θ(g) s'il existe des constantes c_1, c_2, strictement positives et n_0 telles que c1*g(x) ≤ f(x) ≤ c2*g(x) pour tout x ≥ n_0



Combien de temps pour traiter un problème ?

Taille	$\log_2(n)$	n	$n\log_2(n)$	n^2	2^n
10	0.003 ms	0.01 ms	0.03 ms	0.1 ms	1 ms
100	0.006 ms	0.1 ms	0.6 ms	10 ms	10^{14} siècles
1000	0.01 ms	1 ms	10 ms	1 s	
10^4	0.013 ms	10 ms	0.1 s	100 s	
10^5	0.016 ms	100 ms	1.6 s	3 heures	
10^6	0.02 ms	1 s	20 s	10 jours	

pour une machine qui effectue 10^6 traitements par seconde

Complexité des algorithmes

Quel problème peut-on traiter en une seconde ?

nTs	2^n	n^2	$n \log_2(n)$	n	$\log_2(n)$
10^6	20	1000	63000	10^6	10^{300000}
10^7	23	3162	600000	10^7	$10^{3000000}$
10^9	30	31000	4.10^7	10^9	
10^{12}	40	10^6	3.10^{10}		

nTs = nombre d'instructions effectuées chaque seconde

- Calculer la complexité de l'algorithme de multiplication de deux matrices (pour simplifier on prendra deux matrices carrées de taille $n \times n$).
- Soit A, B deux matrices de taille $n \times n$.
- L'algorithme suivant calcule les coefficients c_{ij} de la matrice $C = A \times B$.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$