Algorithmique et structures de données

Julien BERNARD

Université de Franche-Comté – UFR Sciences et Technique
Licence Informatique – 2è année

2013 – 2014

Première partie

Généralités

Plan de ce cours

Plan

1. Introduction
   - À propos de votre enseignant
   - À propos du cours Algorithmique
2. Complexité et Algorithmique
   - Définitions
   - Outils mathématiques
   - Rappels mathématiques

Votre enseignant
Qui suis-je ?

Qui suis-je ?

Julien BERNARD, Maître de Conférence (enseignant-chercheur)
`julien.bernard@univ-fcomte.fr`, Bureau 426C

Enseignement

- Responsable du semestre 5 de la licence Informatique
- Cours : Informatique (L1), Système (L2), Algorithmique et structure de données (L2), Sécurité (L3), Algorithmique Distribuée (M2)
- À distance : Communication dans les systèmes distribués (M2)

Recherche

Optimisation de la communication dans les réseaux de capteurs

Plan

1. Introduction
   - À propos de votre enseignant
   - À propos du cours Algorithmique
2. Complexité et Algorithmique
   - Définitions
   - Outils mathématiques
   - Rappels mathématiques

UE Algorithmique
Organisation

Équipe pédagogique
- Julien BERNARD : Cours (`julien.bernard@univ-fcomte.fr`)
- Anne HEAM : TD, TP (`anne.heam@univ-fcomte.fr`)
- Sébastien CHIPEAUX : TP (`sebastien.chipeaux@femto-st.fr`)

Volume
- Cours : 12 x 1h30, mardi 11h00, amphi C
- TD : 14 x 1h30, vendredi 9h30 (Gr. 2) et 11h00 (Gr. 1)
- TP : 14 x 1h30, lundi 8h00 (A), 11h00 (C) et mardi 8h00 (B)

Évaluation
- 2 devoirs surveillés
- des exercices, un mini-projet, un projet en TP

UE Algorithmique
Comment ça marche ?

Mode d'emploi
1. Cours en ligne (si possible avant le CM)
2. Prenez des notes ! Posez des questions !
3. Le TD n'est pas l'application du cours !
4. Comprendre plutôt qu'apprendre
5. Le but de cette UE n'est pas d'avoir une note !

Niveau d'importance des transparents

	trivial	pour votre culture
$\star$	intéressant	pour votre compréhension
$\star\star$	important	pour votre savoir
$\star\star\star$	vital	pour votre survie

Note : les contrôles portent sur *tous* les transparents !

UE Algorithmique
Contenu pédagogique

Objectif
Acquérir les notions d'algorithmique liées aux structures de données récursives ainsi que les bases de l'analyse d'algorithmes

- Complexité algorithmique
- Pointeurs
- Listes chaînées
- Tris
- Arbres
- Graphes

UE Système
Bibliographie

Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein.
Algorithmique.
3è édition, 2010, Dunod

Donald Knuth.
The Art of Computer Programming.

Plan

Historique ⋆

Qu'est-ce qui est calculable ?

- Première moitié du XX^{e} siècle : recherche de la définition de *calcul*
- K. Gödel, A. Church, A. Turing
- Existence d'une solution $\not\Rightarrow$ calcul effectif d'une solution

→ Qu'est-ce qui est calculable ?

Exemple (Existence sans calcul effectif)

- Racines d'un polynôme quelconque

Qu'est-ce qu'un problème ? ★★★

Définition (Problème)

Un *problème* en informatique est constitué de données sous une certaine forme et d'une question portant sur ces données.

Exemples

Problème *Pâte à crêpes*

- **Données** : 250g de farine, 0,5L de lait, 2 œufs, 2g de sel.
- **Question** : Comment faire une pâte à crêpes ?

Problème *Divisibilité par 10*

- **Données** : $n \in \mathbb{N}$
- **Question** : n est-il divisible par 10 ?

Instance d'un problème ★★★

Définition (Instance d'un problème)

Une *instance d'un problème* est composée d'une valeur pour chaque donnée du problème.

Exemple

1. 10 est une instance du problème «Divisibilité»
2. $(5 \times 3 + 4)$ est une autre instance du même problème

Algorithme ★★★

Définition

Un *algorithme* est une méthode indiquant sans ambiguïté une suite finie d'actions mécaniques à effectuer pour trouver la réponse à un problème.

Précisions sur cette définition

- «sans ambiguïté» exprime le fait que tout le monde comprend la méthode de la même façon.
  - Contre-exemple : «Saler à votre convenance»
- «mécanique» signifie qu'il ne fait pas appel à l'intelligence ou à la réflexion.

Exemple (Un algorithme pour le problème «Divisibilité»)

1. Déterminer le reste r de la division de n par 10.
2. Si $r = 0$, n est divisble par 10 sinon n n'est pas divisble par 10.

Programme ★★★

Définition

Un *programme* désigne la traduction d'un algorithme dans un langage de programmation.

Exemple (Une fonction pour le problème «Divisibilité»)

```
public class DivisiblePar10 {
  public static void main(String[] args) {
    int n = Clavier.saisirInt();
    int r = n % 10;
    if (r == 0) {
      Ecran.afficher(n, " est divisible par 10\n");
    } else {
      Ecran.afficher(n, " n'est pas divisible par 10\n");
    }
  }
}
```


Existe-il un algorithme pour chaque problème ? ★★

Existe-il un algorithme pour chaque problème ?

La réponse est **non** ! Quelques exemples :

- Trouver les prochains numéros du Loto
- Trouver un pavage

Exemple (Définition du problème de pavage)

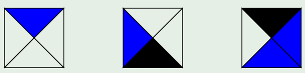
Problème *Pavage*

Données : un ensemble T fini de tuiles carrées dont les bords sont colorés et dont l'orientation est fixée.

Question : Peut-on paver n'importe quelle surface avec des tuiles ayant uniquement des motifs appartenant à T de façon à ce que les couleurs de deux arrêtes de tuiles qui se touchent soient les mêmes ?

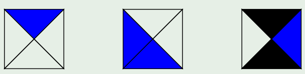
Problème de pavage ★

Exemple (Une instance avec une solution)



(1) (2) (3)

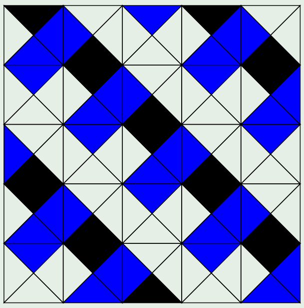
Exemple (Une instance sans solution)



(1) (2) (3)

Problème de pavage ⋆

Exemple (Solution pour la première instance)



Problème indécidable ⋆⋆⋆

Inexistence d'algorithme pour un problème

Que signifie exactement qu'aucun algorithme n'existe pour un problème donné ? Cela veut dire qu'il n'existe pas d'algorithme qui réponde à la question pour **n'importe quelle instance** du problème, c'est-à-dire que pour tout algorithme, il existe une donnée du problème telle que :

- soit l'algorithme ne s'arrête pas ;
- soit il donne une réponse fausse.

Définition (Indécidabilité)

Un problème pour lequel aucun algorithme n'existe est dit *indécidable*.

Tous les algorithmes sont-ils utilisables ? ⋆

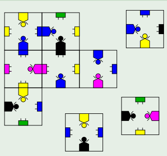
Exemple

Problème *Puzzle des singes*

Données : un ensemble de cartes carrées dont l'orientation est donnée et dont les cotés représentent le bas ou le haut d'un singe

Question : Peut-on arranger ces cartes afin de faire un grand carré tel que les moitiés se correspondent ?

Exemple (Une instance du problème des singes)



Tous les algorithmes sont-ils utilisables ? ⋆

Exemple (Algorithme naïf pour résoudre le puzzle des singes)

1. Tester toutes les combinaisons de cartes jusqu'à en trouver une qui convienne ou à avoir épuisé toutes les possibilités.

→ Étant donné que le nombre de combinaisons de cartes est fini, cet algorithme s'arrête.

Analyse de l'algorithme

Supposons qu'on ait 25 cartes (soit un carré de 5×5), on a donc 25! combinaisons possibles. Si un ordinateur teste un milliard de combinaisons à la seconde, il faudra 490 millions d'années !

→ Cet algorithme est inutilisable !

Problème traitable ⋆⋆⋆

Définition (Problème traitable)

Un problème est dit *traitable* s'il existe un algorithme utilisable pour le résoudre.

Traitable et non-traitable

Un problème peut être non-traitable si les algorithmes pour le résoudre :

- prennent trop de temps ;
- utilisent trop de mémoire.

Si un problème est non-traitable, on peut chercher des algorithmes :

- qui donnent une réponse approchée de la question ;
- qui ne répondent pas toujours.

Complexité et algorithmique ⋆⋆⋆

Définition (Complexité algorithmique)

La *complexité algorithmique* (ou *coût*) est la mesure de l'efficacité d'un algorithme , c'est-à-dire :

- son temps d'exécution ;
- la place mémoire utilisée.

La complexité algorithmique permet de comparer deux algorithmes qui résolvent le même problème.

Définition (Algorithmique)

L'*algorithmique* est l'étude des méthodes pour améliorer la complexité des algorithmes.

Plan

Notations de Landau ★★★

Généralités

Notations de Landau

Les notations de Landau permettent de comparer des fonctions asymptotiquement, c'est-à-dire connaître leur comportement pour des n très grand.

Notation	Signification
$f = O(g)$	f est bornée par g
$f = \Theta(g)$	f est du même ordre que g
$f = o(g)$	f est dominée par g

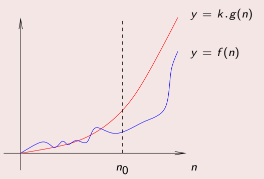
Notations de Landau ★★★

f est bornée par g

Définition (f est bornée par g)

On dit que f est *bornée* par g, et on note $f = O(g)$ si :

$$\exists k > 0, \exists n_0 \in \mathbb{N}, \forall n > n_0, |f(n)| \leq k \cdot |g(n)|$$



Définition intuitive

Pour les grandes valeurs de n, $f(n)$ ne dépasse pas $k \cdot g(n)$.

Notations de Landau

f est bornée par *g* ⋆

Exemples

- $n = O(n)$ avec $n_0 = 0$ et $k = 1$
- $42 \cdot n = O(n)$ avec $n_0 = 0$ et $k = 42$
- $n = O(n^2)$ avec $n_0 = 0$ et $k = 1$
- $n + 3 \cdot n^2 + 4 \cdot n^5 = O(n^5)$
- $42 = O(1)$
- $sin(n) = O(1)$

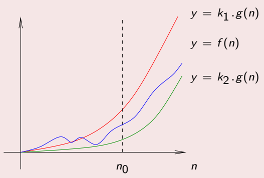
Notations de Landau

f est du même ordre que *g* ⋆⋆⋆

Définition (f est du même ordre que g)

On dit que f est *du même ordre* que g, et on note $f = \Theta(g)$ si :

$$\exists k_1, k_2 > 0, \exists n_0 \in \mathbb{N}, \forall n > n_0, k_1 \cdot |g(n)| \leq |f(n)| \leq k_2 \cdot |g(n)|$$



Définition intuitive

Pour les grandes valeurs de n, $f(n)$ est encadrée par $k_1 \cdot g(n)$ et $k_2 \cdot g(n)$. Ou dit autrement, $f = O(g)$ et $g = O(f)$.

Notations de Landau

f est du même ordre que *g* ⋆

Exemples

- $n = \Theta(n)$ avec $n_0 = 0$ et $k = 1$
- $42 \cdot n = \Theta(n)$ avec $n_0 = 0$ et $k = 42$
- $n + 3 \cdot n^2 + 4 \cdot n^5 = \Theta(n^5)$
- $42 = \Theta(1)$
- $2 + sin(n) = \Theta(1)$

Notations de Landau

f est dominée par *g* ★★★

Définition (f est dominée par g)

On dit que f est *dominée* par g, et on note $f = o(g)$ si :

$$\forall \varepsilon > 0, \exists n_0 \in \mathbb{N}, \forall n > n_0, |f(n)| \leq \varepsilon \cdot |g(n)|$$

Définition intuitive

Pour ε aussi petit qu'on veut, et pour des grandes valeurs de n, $f(n)$ ne dépasse pas $\varepsilon \cdot g(n)$. Dit autrement, pour des grandes valeurs de n, $f(n)$ est tout petit par rapport à $g(n)$.

Définition équivalente

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

Notations de Landau

f est dominée par *g* ★

Exemples

- $42 = o(\log n)$
- $\log n = o(n)$
- $n = o(n^2)$
- $42 \cdot n = o(n^2)$
- $n^2 = o(2^n)$
- $2^n = o(n!)$

Notations de Landau

Notations alternatives ★

Définition (f borne g)

On dit que f borne g, et on note $f = \Omega(g)$, si g est bornée par f, c'est-à-dire si $g = O(f)$.

Définition (f domine g)

On dit que f domine g, et on note $f = \omega(g)$, si g est dominée par f, c'est-à-dire si $g = o(f)$.

Définition (f est équivalente à g)

On dit que f est équivalente à g, et on note $f \sim g$, si $f = g + o(g)$.

Notations de Landau

Utilisation pratique

★★

Utilisations pratiques

- Lorsqu'on aura une fonction f à étudier, on cherchera à trouver :
  1. une fonction simple g telle que $f = \Theta(g)$;
  2. à défaut, une fonction h telle que $f = O(h)$.
- Lorsqu'on aura à comparer deux fonctions f et g, on cherchera à montrer :
  - soit $f = o(g)$ (ou $f = \omega(g)$)
  - soit $f = \Theta(g)$

Échelle de comparaison

★★★

Échelle de comparaison

Fonction	Nom
$O(1)$	constante
$O(\log n)$	logarithmique
$O((\log n)^c)$	polylogarithmique
$O(n)$	linéaire
$O(n \log n)$	log-linéaire
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(n^c)$	polynomiale
$O(c^n)$	exponentielle
$O(n!)$	factorielle

Plan

Logarithme et exponentielle ★★

Logarithme et exponentielle

- $a^{b+c} = a^b \times a^c$
- $a^{b\times c} = \left(a^b\right)^c$
- $\ln(a \times b) = \ln a + \ln b$
- $\ln(a^b) = b \times \ln a$
- $\log_b a = \frac{\ln a}{\ln b}$
- $a^b = e^{b \ln a}$

Formules utiles ★★

Formules utiles

- Somme des premiers entiers

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

- Somme des premiers carrés

$$\sum_{i=1}^{n} i^2 = \frac{(2n+1)(n+1)n}{6} = O(n^3)$$

Formule de Stirling ★★

Formule de Stirling

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Deuxième partie

Complexité algorithmique

Plan de ce cours

3. Complexité algorithmique
   - Définitions
   - Calcul pratique
   - Cas des fonctions récursives

Plan

3. Complexité algorithmique
   - Définitions
   - Calcul pratique
   - Cas des fonctions récursives

Opération fondamentale ⋆⋆⋆

Définition (Opération fondamentale)

Une *opération fondamentale* pour un problème est une opération dont le temps d'exécution pour un algorithme résolvant ce problème est proportionnel au nombre de ces opérations.

Exemples (Opérations fondamentales)

- pour la recherche d'un élément dans un tableau, la *comparaison* entre cet élément et les éléments du tableau
- pour ajouter deux matrices, l'*addition*

Définition (Nombre d'opérations fondamentale)

Pour un algorithme A, une opération fondamentale o et une instance ω du problème, on définit $\mathcal{N}_o(A, \omega)$ comme le nombre d'opérations o lors de l'exécution de A sur ω. On omettra o et ω si le contexte est clair.

Complexité algorithmique ⋆⋆⋆

Définition (Complexité en pire cas)

La *complexité en pire cas* pour un algorithme A sur une instance ω de taille n est définie par :

$$\mathcal{C}_{\text{worst}}(n) \stackrel{\text{def}}{=} \max_{|\omega|=n} \{\mathcal{N}(A, \omega)\}$$

Définition (Complexité en moyenne)

La *complexité en moyenne* pour un algorithme A sur une instance ω de taille n est définie par :

$$\mathcal{C}_{\text{avg}}(n) \stackrel{\text{def}}{=} \frac{\sum_{|\omega|=n} \mathcal{N}(A, \omega)}{\sum_{|\omega|=n} 1}$$

Plan

En pratique ⋆⋆

Comment calculer la complexité ?

Pour un algorithme ou un ensemble d'algorithmes qui résolvent le même problème, on va :

1. Choisir une opération fondamentale o
2. Déterminer ce que représente la taille n d'une instance ω
3. Compter le nombre $\mathcal{N}$ d'opérations fondamentales de l'algorithme

Exemple (Nombre d'opérations d'une expression simple)

Dans la suite de cette section, on prend comme opération fondamentale l'addition. Soit l'expression E simple :

$1 + 2$

Alors, le nombre d'opérations fondamentales est :

$$\mathcal{N}(E) = 1 = O(1)$$

Cas d'une instruction ⋆⋆

Cas d'une instruction

Soit l'instruction I unique :

instruction

On compte le nombre d'opérations fondamentales autant de fois qu'elle apparaît dans l'instruction :

$$\mathcal{N}(I) = \mathcal{N}(\text{instruction})$$

Exemple (Nombre d'opérations d'une instruction)

Soit l'instruction I :

$x \leftarrow 1 + 2 + 3 + 4$

Alors, le nombre d'opérations fondamentales est :

$$\mathcal{N}(I) = 3 = O(1)$$

Cas d'une séquence d'instructions ⋆⋆

Cas d'une séquence d'instructions

Soit la séquence S d'instructions :

instruction$_1$
instruction$_2$
...
instruction$_i$
...
instruction$_k$

Le nombre d'opérations fondamentales est :

$$\mathcal{N}(S) = \sum_{i=1}^{k} \mathcal{N}(\text{instruction}_i)$$

Cas d'une séquence d'instructions ⋆

Exemple (Nombre d'opérations d'une séquence)

Soit la séquence S d'instructions :

$x \leftarrow 3$
$y \leftarrow 4 + x$
$z \leftarrow 5 + y + x$
$d \leftarrow x * x + y * y + z * z$

Alors, le nombre d'opérations fondamentales est :

$$\mathcal{N}(S) = 0 + 1 + 2 + 2 = 5 = O(1)$$

Cas d'une condition ⋆⋆

Cas d'une condition

Soit la séquence S d'instructions :

if expression **then**
    séquence$_1$
else
    séquence$_2$
end if

Le nombre d'opérations fondamentales est :

$$\mathcal{N}(S) \leq \mathcal{N}(\text{expression}) + \max\{\mathcal{N}(\text{séquence}_1), \mathcal{N}(\text{séquence}_2)\}$$

Cas d'une condition ⋆

Exemple (Nombre d'opérations d'une condition)

Soit la séquence S d'instructions :

if $a + b + c < 10$ **then**
    $b \leftarrow 5$
else
    $c \leftarrow a + b + 2$
end if

Alors, le nombre d'opérations fondamentales est :

$$\mathcal{N}(S) \leq 2 + \max\{0, 2\} = 4 = O(1)$$

Cas d'une boucle

Cas d'une boucle `while`

★★

Cas d'une boucle `while`

Soit la séquence S d'instructions :

while expression **do**
  séquence
end while

Si le nombre de passage dans la boucle est k, le nombre d'opérations fondamentales est :

$$\mathcal{N}(S) = (k+1) * \mathcal{N}(\text{expression}) + k * \mathcal{N}(\text{séquence})$$

Attention ! Déterminer k peut être difficile. On essaiera alors de déterminer un majorant $k' \geq k$ de sorte que :

$$\mathcal{N}(S) \leq (k'+1) * \mathcal{N}(\text{expression}) + k' * \mathcal{N}(\text{séquence})$$

Cas d'une boucle

Cas d'une boucle `while`

★

Exemple (Nombre d'opérations d'une boucle `while` (1))

Soit la séquence S d'instructions :

$a \leftarrow 0$
while $a < 10$ **do**
  $a \leftarrow a + 2$
end while

Le nombre de boucle est 5 et le nombre d'opérations fondamentales est :

$$\mathcal{N}(S) \leq 0 + 5 * 1 = 5 = O(1)$$

Cas d'une boucle

Cas d'une boucle `while`

★

Exemple (Nombre d'opérations d'une boucle `while` (2))

Soit la séquence S d'instructions :

$x \leftarrow 1$
$y \leftarrow x$
while $x < n$ **do**
  $x \leftarrow x + x$
  $y \leftarrow y + x$
end while

Le nombre de boucle est de $\lfloor \log_2(n) \rfloor$ et le nombre d'opérations fondamentales est :

$$\mathcal{N}(S) \leq 0 + \lfloor \log_2(n) \rfloor * 2 = O(\log n)$$

Cas d'une boucle

Cas d'une boucle `for`

★★

Cas d'une boucle `for`

Soit la séquence S d'instructions :

for i **from** a **to** b **do**
  séquence
end for

Si le nombre de passage dans la boucle est $k = b - a + 1$, le nombre d'opérations fondamentales est :

$$\mathcal{N}(S) = k * \mathcal{N}(\text{séquence})$$

Cas d'une boucle

Cas d'une boucle `for`

★

Exemple (Nombre d'opérations d'une boucle `for` (1))

Soit la séquence S d'instructions :

for i **from** 1 **to** 4 **do**
  $x \leftarrow x + i$
end for

Le nombre de boucle est de $4 - 1 + 1 = 4$ et le nombre d'opérations fondamentales est :

$$\mathcal{N}(S) = 4 * 1 = 4 = O(1)$$

Cas d'une boucle

Cas d'une boucle `for`

★

Exemple (Nombre d'opérations d'une boucle `for` (2))

Soit la séquence S d'instructions :

for i **from** 1 **to** n **do**
  $x \leftarrow x + i + 3$
end for

Le nombre de boucle est de $n - 1 + 1 = n$ et le nombre d'opérations fondamentales est :

$$\mathcal{N}(S) = n * 2 = O(n)$$

Cas d'une boucle

Cas d'une boucle for

★

Exemple (Nombre d'opérations d'une boucle for (3))

Soit la séquence S d'instructions :

for i **from** 1 **to** 4 **do**
  for j **from** 1 **to** 3 **do**
    $x \leftarrow x + j$
  end for
end for

Le nombre d'opérations fondamentales de la séquence interne S' est :

$$\mathcal{N}(S') = 3 * 1 = 3$$

Le nombre d'opération fondamentales est :

$$\mathcal{N}(S) = 4 * \mathcal{N}(S') = 4 * 3 = 12 = O(1)$$

Cas d'une fonction non-récursive

★★

Cas d'une fonction non-récursive

Soit la fonction F :

function F(paramètres)
  séquence
end function

Le nombre d'opérations fondamentales est :

$$\mathcal{N}(\mathrm{F}) = \mathcal{N}(\text{séquence})$$

Cas d'une fonction non-récursive

★

Exemple (Nombre d'opération d'une fonction non-récursive)

Soit la fonction $\textsc{Double}$:

function $\textsc{Double}(a)$
  return $a + a$
end function

Alors, le nombre d'opérations fondamentales est :

$$\mathcal{N}(\textsc{Double}) = 1 = O(1)$$

Plan

3. Complexité algorithmique
   - Définitions
   - Calcul pratique
   - Cas des fonctions récursives

Cas d'une fonction récursive simple ⋆⋆

Cas d'une fonction récursive simple

Soit la fonction F :

function $\mathrm{F}(n)$
  if $n = 0$ **then**
    séq$_1$
    return
  end if
  séq$_2$
  $\mathrm{F}(n-1)$
  séq$_3$
end function

La complexité de F est définie par :

$$\begin{cases} \mathcal{C}(0) = \mathcal{N}(\text{séq}_1) \\ \mathcal{C}(n) = \mathcal{N}(\text{séq}_2) + \mathcal{C}(n-1) + \mathcal{N}(\text{séq}_3) \end{cases}$$

On peut démontrer (par récurrence) que la complexité de F est :

$$\mathcal{C}(n) = \mathcal{N}(\text{séq}_1) + n \times (\mathcal{N}(\text{séq}_2) + \mathcal{N}(\text{séq}_3))$$

Cas d'une fonction récursive simple ⋆

Exemple (Complexité de factorielle (1/2))

Problème *Factorielle*

Données : $n \in \mathbb{N}$

Résultat : $n! = 1 \times 2 \times \ldots \times n$

Soit l'algorithme suivant qui résout ce problème :

function $\textsc{Factorielle}(n)$
  if n $= 0$ **then**
    return 1
  end if
  return $n \times \textsc{Factorielle}(n-1)$
end function

Cas d'une fonction récursive simple ⋆

Exemple (Complexité de factorielle (2/2))

L'opération fondamentale est ici la multiplication $\times$. La complexité de cet algorithme est définie par :

$$\begin{cases} \mathcal{C}(0) = 0 \\ \mathcal{C}(n) = 1 + \mathcal{C}(n-1) \end{cases}$$

Donc, la complexité de cet algorithme est :

$$\mathcal{C}(n) = n = \Theta(n)$$

Cas général d'une fonction récursive ⋆⋆⋆

Cas général d'une fonction récursive

Le cas général traite des fonctions récursives de la forme suivante :

```
function F(n)
    if n = 0 then
        return
    end if
    séquence // division
    F(n/k) // 1er appel récursif
    ...
    F(n/k) // ae appel récursif
    séquence // fusion
end function
```

avec :

- $a \geq 1$, constante
- $k > 1$, constante
- $\mathcal{N}(\text{séquence}) = f(n)$

Alors, la complexité de cet algorithme est défini par :

$$\mathcal{C}(n) = a \times \mathcal{C}\left(\frac{n}{k}\right) + f(n)$$

Le théorème suivant donne une mesure asymptotique de $\mathcal{C}(n)$.

Théorème Diviser pour régner (*Master Theorem*) ⋆⋆⋆⋆

Théorème (Théorème Diviser pour régner)

Si $\mathcal{C}(n) = a \times \mathcal{C}\left(\frac{n}{k}\right) + f(n)$ *alors :*

1. *Si* $f(n) = O\left(n^{\log_k (a-\varepsilon)}\right)$ *avec* $\varepsilon > 0$, *alors :*
$$\mathcal{C}(n) = \Theta\left(n^{\log_k a}\right)$$
2. *Si* $f(n) = \Theta\left(n^{\log_k a}\right)$, *alors :*
$$\mathcal{C}(n) = \Theta\left(n^{\log_k a} \log n\right)$$
3. *Si* $f(n) = \Omega\left(n^{\log_k (a+\varepsilon)}\right)$ *avec* $\varepsilon > 0$,
et si $\exists c < 1, a \times f\left(\frac{n}{k}\right) \leq c \times f(n)$ *pour n assez grand, alors :*
$$\mathcal{C}(n) = \Theta(f(n))$$

Théorème Diviser pour régner

Cas où $f(n) = O(n)$

★★★

Théorème (Théorème Diviser pour régner dans le cas où $f(n) = O(n)$)

Si $\mathcal{C}(n) = a \times \mathcal{C}\left(\frac{n}{k}\right) + O(n)$ alors :

1. *Si $a > k$, alors $\mathcal{C}(n) = O(n^{\log_k a})$*
2. *Si $a = k$, alors $\mathcal{C}(n) = O(n \log n)$*
3. *Si $a < k$, alors $\mathcal{C}(n) = O(n)$*

Cas d'utilisation

Dans la pratique, c'est cette version du théorème qu'on utilisera le plus souvent. Elle découle immédiatement du théorème dans sa version générale.

Théorème Diviser pour régner

Cas où $a = 1$

★★★

Théorème (Théorème Diviser pour régner dans le cas où $a = 1$)

Si $\mathcal{C}(n) = \mathcal{C}\left(\frac{n}{k}\right) + f(n)$ alors :

$$\mathcal{C}(n) = \mathcal{C}(1) + \sum_{i=1}^{\log_k n} f\left(k^i\right)$$

Cas d'utilisation

C'est l'autre grand cas d'utilisation pratique du théorème général. En particulier, quand $f(n) = O(1)$, alors :

$$\mathcal{C}(n) = O(\log n)$$

Exemples classiques

★★

Exemples (Application du théorème Diviser pour régner)

1. Si $\mathcal{C}(n) = \mathcal{C}\left(\frac{n}{2}\right) + O(1)$ alors :
$$\mathcal{C}(n) = O(\log n)$$
2. Si $\mathcal{C}(n) = 2 \times \mathcal{C}\left(\frac{n}{2}\right) + O(1)$ alors :
$$\mathcal{C}(n) = O(n)$$
3. Si $\mathcal{C}(n) = 2 \times \mathcal{C}\left(\frac{n}{2}\right) + O(n)$ alors :
$$\mathcal{C}(n) = O(n \log n)$$
4. Si $\mathcal{C}(n) = 3 \times \mathcal{C}\left(\frac{n}{2}\right) + O(n)$ alors :
$$\mathcal{C}(n) = O(n^{\log_2 3})$$

Troisième partie

Pointeurs et tableaux

Plan de ce cours

Plan

Définition ⋆⋆

Définition (Pointeur)

Quel que soit le type $\mathcal{T}$, on peut définir un type «*pointeur* sur $\mathcal{T}$». Une variable de type «pointeur sur $\mathcal{T}$» peut contenir l'adresse d'une variable (ou plus généralement d'un objet en mémoire) de type $\mathcal{T}$.

Remarque

Le type «pointeur sur $\mathcal{T}$» étant un type comme les autres, il est également possible de définir un type «pointeur sur pointeur sur $\mathcal{T}$» et ainsi de suite.

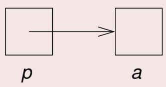
Vocabulaire et représentation ⋆⋆

Vocabulaire

Si la valeur d'un pointeur p est l'adresse d'une variable a, alors :

- on dit que p *pointe* sur a
- la valeur de a est appelé le *contenu* de p

Représentation générique



Représentation ⋆

Code source

Exemple (Code source en C)

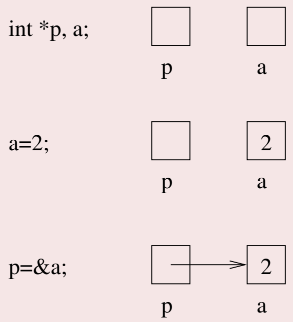
```
int *p, a; // p est un ? et a est un ?
a = 2;
p = &a;
```

Représentation
Représentation générique

Exemple (Code source en C)

```c
int *p, a;
a = 2;
p = &a;
```

Représentation générique

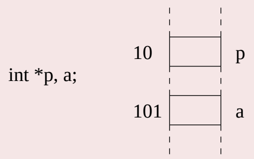


Représentation
Représentation mémoire

Exemple (Code source en C)

```c
int *p, a;
a = 2;
p = &a;
```

Représentation mémoire

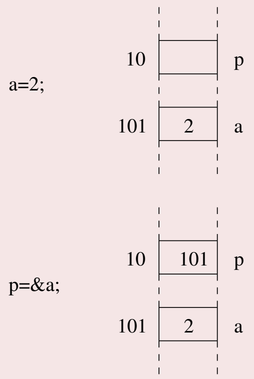


Représentation
Représentation mémoire

Exemple (Code source en C)

```c
int *p, a;
a = 2;
p = &a;
```

Représentation mémoire



Plan

4 Généralités
- Pointeurs
- Tableaux

5 Algorithmes
- Algorithmes sur les tableaux
- Algorithmes sur les chaînes de caractères
- Tableaux dynamiques

Définition et représentation ⋆⋆

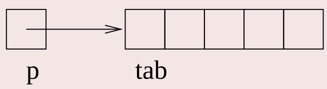
Définition (Tableau)

Quel que soit le type $\mathcal{T}$, on peut définir un type «*tableau* de $\mathcal{T}$». Une variable de type «tableau de $\mathcal{T}$» est un pointeur vers le premier élément parmi n qui sont rangés de manière contigus en mémoire.

Exemple (Code source en C)

```c
int *p, tab[5];
p = tab; // equivalent a : p = &tab[0]
```

Représentation générique

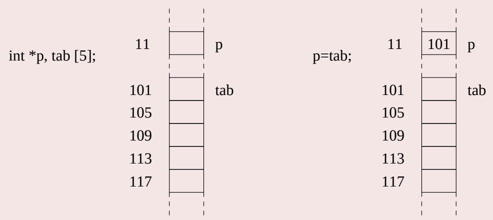


Représentation mémoire ⋆

Exemple (Code source en C)

```c
int *p, tab[5];
p = tab; // equivalent a : p = &tab[0]
```

Représentation mémoire



Plan

4 Généralités
- Pointeurs
- Tableaux

5 Algorithmes
- Algorithmes sur les tableaux
- Algorithmes sur les chaînes de caractères
- Tableaux dynamiques

Tableaux ★★

Suppositions

On considère ici des tableaux :
- dont la taille n est connue et constante
- indicé à partir de 0 jusqu'à $n-1$

Opération de base sur les tableaux

- *accès aléatoire* à l'élément d'indice i :

  t[i]

  Complexité : $O(1)$

Recherche dans un tableau ★★

Définition du problème

Problème *Recherche d'un élément dans un tableau*

Données : un tableau t de taille n et un élément e

Résultat : l'indice de l'élément e dans le tableau t ou n si l'élément n'est pas dans le tableau

Algorithme

function SEARCH(t, n, e)
  $i \leftarrow 0$
  while $i < n$ && $t[i] \neq e$ **do**
    $i \leftarrow i + 1$
  end while
  return i
end function

Recherche dans un tableau ⋆⋆⋆

Complexité

L'opération fondamentale est la comparaison ($\neq$). Plusieurs cas se présentent :

- Pire cas : l'élément n'est pas présent dans le tableau, dans ce cas, on effectue n comparaisons.

$$\mathcal{C}_{\text{worst}}(n) = n = O(n)$$

- En moyenne : l'élément est dans le tableau, il se situe en moyenne à l'indice $\frac{n}{2}$, dans ce cas, on effectue $\frac{n}{2}$ comparaisons.

$$\mathcal{C}_{\text{avg}}(n) = \frac{n}{2} = O(n)$$

Recherche dans un tableau trié ⋆⋆

Définition du problème

Problème *Recherche d'un élément dans un tableau trié*

Données : un tableau t de taille n trié par ordre croissant et un élément e

Résultat : l'indice de l'élément e dans le tableau t ou n si l'élément n'est pas dans le tableau

Algorithme de recherche dichotomique

On va utiliser l'algorithme de *recherche dichotomique*. On introduit deux variables a et b qui sont les indices de début et de fin de la recherche, plus précisément l'élément e se trouve dans l'intervalle $[a; b[$.

Recherche dans un tableau trié ⋆⋆⋆

Algorithme de recherche dichotomique

```
function BinarySearch(t, n, e, a, b)
    if a = b then
        return n
    end if
    m ← (a+b)/2
    if t[m] = e then
        return m
    else if e < t[m] then
        return BinarySearch(t, n, e, a, m)
    else
        return BinarySearch(t, n, e, m + 1, b)
    end if
end function
```

Recherche dans un tableau trié ★★★

Algorithme général

function SEARCHSORTED(t, n, e)
    return BINARYSEARCH(t, n, e, 0, n)
end function

Complexité

L'opération fondamentale est la comparaison ($<$, $>$ et $=$). Pour un tableau de taille n (c'est-à-dire $b - a = n$), on a :

$$\mathcal{C}(n) = \mathcal{C}\left(\frac{n}{2}\right) + O(1)$$

Grâce au Théorème Diviser pour Régner, on en déduit :

$$\mathcal{C}(n) = O(\log n)$$

Insertion d'un élément dans un tableau ★★

Problème

Problème *Insertion d'un élément dans un tableau*

Données : un tableau t de taille n occupé par $m < n$ éléments et un élément e à insérer à l'indice $j \in [O, m]$

Résultat : le tableau t avec $m + 1$ éléments et l'élément e à l'indice j

Il existe plusieurs variantes de ce problème :

- *Insertion en fin*, c'est-à-dire $j = m$. Dans ce cas, l'algorithme est trivial et sa complexité est en $O(1)$
- *Insertion sans conservation de l'ordre*. Dans ce cas, l'algorithme consiste à placer l'ancien élément d'indice j à l'indice m pour laisser la place à e à l'indice j. La complexité est en $O(1)$.
- *Insertion avec conservation de l'ordre*. C'est cet algorithme là que nous allons voir.

Insertion d'un élément dans un tableau ★★

Algorithme

function INSERTELEMENT(t, m, e, j)
    for i **from** m **to** $j + 1$ **do**
        $t[i] \leftarrow t[i-1]$
    end for
    $t[j] \leftarrow e$
end function

Complexité

L'opération fondamentale est l'affectation ($\leftarrow$). La complexité de cet algorithme est de $m - j$ affectations. En moyenne, $j = \frac{m}{2}$, donc :

$$\mathcal{C}(m) = m - \frac{m}{2} = \frac{m}{2} = O(m)$$

Suppression d'un élément dans un tableau ⋆⋆

Problème

Problème *Suppression d'un élément dans un tableau*

> **Données** : un tableau t de taille n occupé par $m \leq n$ éléments et un élément à supprimer à l'indice $j \in [O, m[$
>
> **Résultat** : le tableau t avec $m - 1$ éléments

Il existe plusieurs variantes de ce problème :

- *Suppression en fin*, c'est-à-dire $j = m - 1$. Dans ce cas, l'algorithme est trivial et sa complexité est en $O(1)$
- *Suppression sans conservation de l'ordre*. Dans ce cas, l'algorithme consiste à placer l'élément d'indice $m - 1$ à l'indice j. La complexité est en $O(1)$.
- *Suppression avec conservation de l'ordre*. C'est cet algorithme là que nous allons voir.

Suppression d'un élément dans un tableau ⋆⋆

Algorithme

```
function DeleteElement(t, m, j)
    for i from j + 1 to m − 1 do
        t[i − 1] ← t[i]
    end for
end function
```

Complexité

L'opération fondamentale est l'affectation (←). La complexité de cet algorithme est de $m - j - 1$ affectations. En moyenne, $j = \frac{m}{2}$, donc :

$$\mathcal{C}(m) = m - \frac{m}{2} = \frac{m}{2} = O(m)$$

Plan

Chaînes de caractères ⋆⋆

Définition (Chaîne de caractères)

Une *chaîne de caractères* est un tableau de caractères dont le dernier élément est 0.

Supposition

Généralement, on ne connaît pas la taille de la chaîne à l'avance. Tous les algorithmes sur les tableaux s'appliquent également aux chaînes de caractères, en veillant à ce que le dernier caractère soit toujours 0.

Taille d'une chaîne de caractères ⋆⋆

Problème

Problème *Taille d'une chaîne de caractères*

Données : une chaîne de caractères s

Résultat : la taille de la chaîne s, c'est-à-dire le nombre de caractères contenus dans la chaîne sans le 0 final

Remarque

La fonction C équivalente est `strlen(3)`

Taille d'une chaîne de caractères ⋆⋆

Algorithme

function LENGTH(s)
    $i \leftarrow 0$
    while $s[i] \neq 0$ **do**
        $i \leftarrow i + 1$
    end while
    return i
end function

Complexité

L'opération fondamentale est la comparaison ($\neq$). La complexité pour une chaîne de taille n est de n comparaisons. Donc :

$$\mathcal{C}(n) = n = O(n)$$

Comparaison de chaînes de caractères ⋆⋆

Problème

Problème *Comparaison de chaînes de caractères*

Données : une chaîne de caractères s et une chaine de caractère u

Résultat : un entier strictement positif/nul/strictement négatif suivant que la chaîne s est inférieure/égale/supérieure à la chaîne u selon l'ordre lexicographique

Remarque

La fonction C équivalente est `strcmp(3)`

Algorithmes | Algorithmes sur les chaînes de caractères

Comparaison de chaînes de caractères ⋆⋆

Algorithme

function COMPARE(s, u)
  $i \leftarrow 0$
  while $s[i] \neq 0$ && $u[i] \neq 0$ **do**
    if $s[i] \neq u[i]$ **then**
      return $s[i] - u[i]$
    end if
  end while
  return $s[i] - u[i]$
end function

Algorithmes | Algorithmes sur les chaînes de caractères

Comparaison de chaînes de caractères ⋆⋆

Complexité

L'opération fondamentale est la comparaison ($\neq$). La complexité pour une chaîne s de taille n et une chaîne u de taille m est :

- Pire cas : les chaînes sont presque égales, sauf le dernier caractère (par exemple : «aaaa» et «aaab»), on fait $\min(n, m)$ comparaisons.

$$\mathcal{C}_{\text{worst}}(n, m) = \min(n, m) = O(\min(n, m))$$

- En moyenne : on échoue à la première lettre, et on fait une seule comparaison.

$$\mathcal{C}_{\text{avg}}(n, m) = 1 = O(1)$$

Recherche d'une sous-chaîne ★★

Problème

Problème *Recherche d'une sous-chaîne*

Données : une chaîne de caractères s dans laquelle on va chercher et une chaine de caractère u que l'on va chercher

Résultat : l'indice j à laquelle se trouve la sous-chaîne u dans la chaîne de caractères s ou -1 en cas d'échec

Remarque

La fonction C équivalente est `strstr(3)`

Recherche d'une sous-chaîne ★★

Algorithme

```
function MATCH(s, u)
    i ← 0
    while s[i] ≠ 0 do
        j ← 0
        while u[j] ≠ 0 && s[i + j] ≠ 0 && s[i + j] = u[j] do
            j ← j + 1
        end while
        if u[j] = 0 then
            return i
        end if
        i ← i + 1
    end while
    return −1
end function
```


Recherche d'une sous-chaîne ★★★

Complexité

L'opération fondamentale est la comparaison $(=)$. La complexité pour une chaîne s de taille n et une chaîne u de taille m est :

- Pire cas : on échoue à la dernière lettre de u à chaque fois (par exemple, si on cherche la chaîne «aaaab» dans la chaîne «aaaaaaaaab»), on fait alors $n \times m$ comparaisons.

$$\mathcal{C}_{\text{worst}}(n, m) = nm = O(nm)$$

- En moyenne : on échoue à la première lettre de u jusqu'à trouver la chaîne (en moyenne au milieu de la chaîne s), alors on fait $\frac{n}{2} + m$ comparaisons.

$$\mathcal{C}_{\text{avg}}(n, m) = \frac{n}{2} + m = O(n + m)$$

Plan

4 Généralités
- Pointeurs
- Tableaux

5 Algorithmes
- Algorithmes sur les tableaux
- Algorithmes sur les chaînes de caractères
- Tableaux dynamiques

Tableau dynamique ★★

Définition

Un *tableau dynamique* est un tableau dont la taille varie suivant les besoins. On le représente par une structure de donnée comprenant trois champs :

- p : un tableau alloué dynamiquement (*pointer*)
- c : la taille en mémoire du tableau (*capacity*)
- s : le nombre d'éléments dans le tableau (*size*)

Usage

Les tableaux dynamiques permettent d'abstraire la gestion de la mémoire des tableaux dans des fonctions. Ils font partie de la bibliothèque standard de nombreux langages. . . mais pas le C.

Insertion en fin d'un tableau dynamique ★★★

Problème

Problème *Insertion en fin d'un tableau dynamique*

Données : un tableau dynamique t et un élément e à insérer

Résultat : le tableau dynamique t avec un élément e en plus à la fin

Remarque importante

Contrairement à l'insertion dans un tableau de taille fixe, il est toujours possible d'insérer un élément dans un tableau dynamique puisqu'au besoin, celui-ci peut grossir. La seule question intéressante est de savoir comment grossir.

Insertion en fin d'un tableau dynamique ⋆⋆

Algorithme naïf

function INSERT(t, e)
  if $t.s = t.c$ **then**
    $t.c \leftarrow t.c + 1$
    $p \leftarrow$ ALLOC($t.c$)
    COPY(p, $t.p$, $t.s$)
    FREE($t.p$)
    $t.p \leftarrow p$
  end if
  $t.p[t.s] = e$
  $t.s \leftarrow t.s + 1$
end function

Insertion en fin d'un tableau dynamique ⋆⋆⋆

Définition (Coût amorti)

On appelle *coût amorti* le coût moyen d'un algorithme sur un grand nombre d'appels successifs à l'algorithme. On utilise le coût amorti quand un algorithme se comporte bien dans la plupart des cas et mal dans quelques cas particuliers.

Complexité

L'opération fondamentale est l'affectation ($\leftarrow$). Dans ce cas, en admettant que la capacité originale soit de 1, à chaque insertion, on fait un appel à COPY (qui est en $O(t.s)$), donc au bout de n appels, on a fait : $1 + 2 + 3 + \ldots + n$ copies d'éléments du tableaux et n ajout de e d'où un coût total en $O(n^2)$. Si on divise par le nombre d'appels, le coût amorti d'une insertion est :

$$\mathcal{C}_{\text{amort}}(n) = O(n)$$

Insertion en fin d'un tableau dynamique ⋆⋆⋆

Algorithme

function INSERT(t, e)
  if $t.s = t.c$ **then**
    $t.c \leftarrow A \times t.c$
    $p \leftarrow$ ALLOC($t.c$)
    COPY(p, $t.p$, $t.s$)
    FREE($t.p$)
    $t.p \leftarrow p$
  end if
  $t.p[t.s] = e$
  $t.s \leftarrow t.s + 1$
end function

où A est une constante avec $A > 1$ (généralement $A = 2$)

Insertion en fin d'un tableau dynamique ⋆⋆⋆

Complexité

Dans ce cas, en admettant que la capacité originale soit de 1, quand on augmente la taille, on la multiplie par A. Donc, au bout de n appels avec $A^k \leq n < A^{k+1}$, on a fait k augmentation de taille, qui représente au total : $1 + A + A^2 + A^3 + \ldots + A^{k+1} = \frac{1-A^{k+2}}{1-A} = O(A^k)$ copies d'éléments du tableaux et n ajout de e. Or, $k = \log_A n$, donc $A^k = n$, donc, pour n éléments insérés, on fait au total $O(n) + n$ copies. Et donc, le coût amorti d'une insertion est :

$$\mathcal{C}_{\text{amort}}(n) = O(1)$$

Insertion en fin d'un tableau dynamique ⋆

Remarques

- Cette stratégie d'allocation s'appelle une *expansion géométrique*.
- L'espace non-utilisé est au maximum de $(A-1)n$ éléments de sorte qu'on peut choisir A proche de 1 pour minimiser l'espace non-utilisé.
- On peut aussi réduire la taille du tableau si l'occupation descend en dessous d'un certain seuil. Il faut alors choisir ce seuil inférieur à $\frac{1}{A}$ pour éviter d'avoir des allocations et désallocations successives.

C'est tout pour le moment. . .

Des questions ?