Algorithmique et structures de données

Julien BERNARD

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

2015 – 2016

Première partie

Introduction et généralités

Plan de ce cours

1. Introduction
   - À propos de votre enseignant
   - À propos du cours Algorithmique

2. Complexité et Algorithmique
   - Définitions
   - Outils mathématiques
   - Rappels mathématiques

Plan

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 1 (Starter) de la licence Informatique
- Cours : Publication web et scientifique (L1), Algorithmique (L2), Système (L2), Sécurité (L3)

Recherche

Optimisation dans les réseaux de capteurs

Plan

UE Algorithmique
Organisation

Équipe pédagogique

- Julien Bernard : CM, TD, TP (julien.bernard@univ-fcomte.fr)
- Mohamed Hadded : TP (mohamed.hadded@univ-fcomte.fr)

Volume

- Cours : 12 x 1h30, jeudi 11h00
- TD : 14 x 1h30, vendredi 9h30 (Gr. 1) et 11h00 (Gr. 2)
- TP : 14 x 1h30

Évaluation

- 2 devoirs surveillés
- (au moins) un projet en TP

UE Algorithmique
Comment ça marche ?

Mode d'emploi

1. Pas de cours en ligne. . . mais les algorithmes importants
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 et tableaux
- Listes chaînées
- Tris
- Arbres
- Graphes

UE Algorithmique
Bibliographie


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


Donald Knuth.
The Art of Computer Programming.

UE Algorithmique

Hommage

Alan Rickman (21/02/1946 – 14/01/2016)

$\rightarrow$ Semestre spécial Severus Rogue !

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

$\rightarrow$ 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)

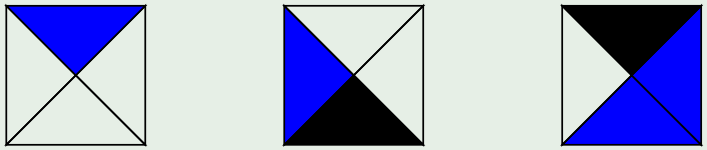
Problème *Pavage*

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 ?

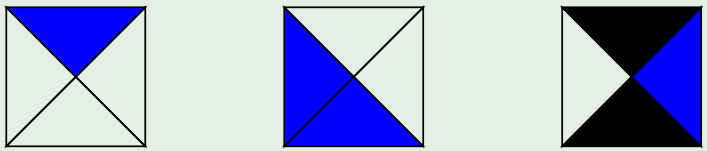
Problème de pavage

Exemple (Une instance avec une solution)



(1) (2) (3)

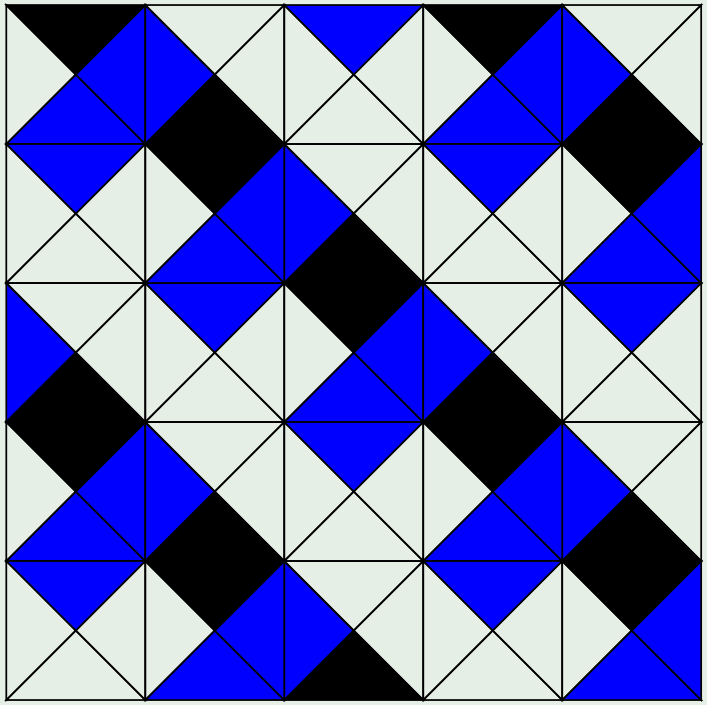
Exemple (Une instance sans solution)



(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 ? ⋆

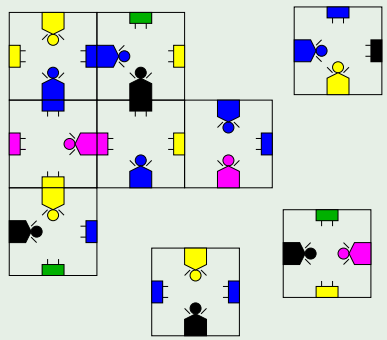
Exemple

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

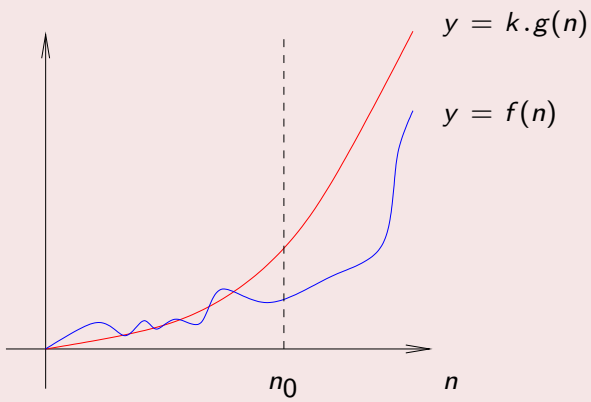
f est bornée par g

★★★

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

Notations de Landau

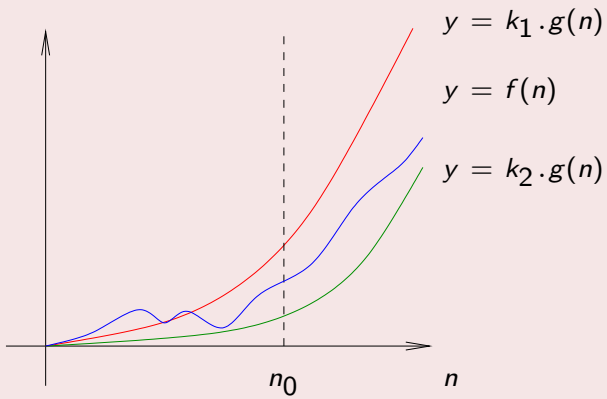
f est du même ordre que g

★★★

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

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

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

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

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

x prend toutes les valeurs des puissances de 2 jusuq'as dépasser n (d'ou log2(n))

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 DOUBLE :

function DOUBLE(a)
    return $a + a$
end function

Alors, le nombre d'opérations fondamentales est :

$$\mathcal{N}(\text{DOUBLE}) = 1 = O(1)$$

Plan

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 FACTORIELLE(n)
    if n = 0 then
        return 1
    end if
    return n × 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 $\mathrm{F}(n)$
  if $n = 0$ **then**
    return
  end if
  séquence // division
  $\mathrm{F}(\frac{n}{b})$ // 1^{er} appel récursif
  ...
  $\mathrm{F}(\frac{n}{b})$ // a^{e} appel récursif
  séquence // fusion
end function

avec :

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

Alors, la complexité de cet algorithme est défini par :

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

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

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

Théorème (Théorème Diviser pour régner)

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

1. *Si* $f(n) = O\left(n^c\right)$ *avec* $c < \log_b a$, *alors :*

$$\mathcal{C}(n) = \Theta\left(n^{\log_b a}\right)$$

2. *Si* $f(n) = \Theta\left(n^c \log^k n\right)$ *avec* $c = \log_b a$, *alors :*

$$\mathcal{C}(n) = \Theta\left(n^c \log^{k+1} n\right)$$

3. *Si* $f(n) = \Omega\left(n^c\right)$ *avec* $c > \log_b a$,
   et si $\exists k < 1, a \times f\left(\frac{n}{b}\right) \leq k \times f(n)$ *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 $\mathcal{C}(n) = a \times \mathcal{C}\left(\frac{n}{b}\right) + O(n)$ *alors :*

1. *Si* $a > b$, *alors* $\mathcal{C}(n) = O(n^{\log_b a})$
2. *Si* $a = b$, *alors* $\mathcal{C}(n) = O(n \log n)$
3. *Si* $a < b$, *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 $\mathcal{C}(n) = \mathcal{C}\left(\frac{n}{b}\right) + f(n)$ *alors :*

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

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

4. Structures de données
   - Introduction
   - Implémentation

5. Pointeurs et tableaux
   - Pointeurs
   - Tableaux

6. Algorithmes sur les tableaux
   - Algorithmes sur les tableaux de taille fixe
   - Algorithmes sur les chaînes de caractères
   - Tableaux dynamiques

Plan

Structures de données

Définition (Structure de données)

Une structure de données est une structure logique destinée à recevoir des données, afin de leur donner une organisation permettant de simplifier leur traitement.

Exemples

- Structures linéaires : tableau, liste
- Structures arborescentes : arbre, graphe

Structures de données

Niveaux de description

1. **structure mathématique** : structure définie mathématiquement, souvent récursivement
2. **structure abstraite** : ensemble d'opérations avec des garanties de complexité (interface) et axiomes pour décrire le comportement de ces opérations
3. **structure logique** : représentation logique de la structure, indépendemment d'un langage de programmation, manipulée par un langage algorithmique
4. **structure réelle** : implémentation concrète de la structure à l'aide d'un langage de programmation

Plan

Implémentation

Conventions

Dans ce cours, les structures de données seront données en langage C. Toutes les fonctions se rapportant à une structure auront un nom préfixé par le nom de la structure et prendront en *premier paramètre* un pointeur sur la structure appelé `self`.

Fonctions

Pour une structure nommée `foo`, on définira obligatoirement :

- `void foo_create(struct foo *self);`
  initialise une structure vide
- `void foo_destroy(struct foo *self);`
  libère toutes les ressources interne de la structure

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;
int a;
a = 2;
p = &a;
```

Déclaration

- p est un pointeur sur un entier
- a est un entier

Représentation

Représentation générique

Exemple (Code source en C)

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

Représentation générique


```
int *p; int a;
```

Représentation
Représentation générique

Exemple (Code source en C)

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

Représentation générique

Représentation

Représentation générique

Exemple (Code source en C)

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

Représentation générique

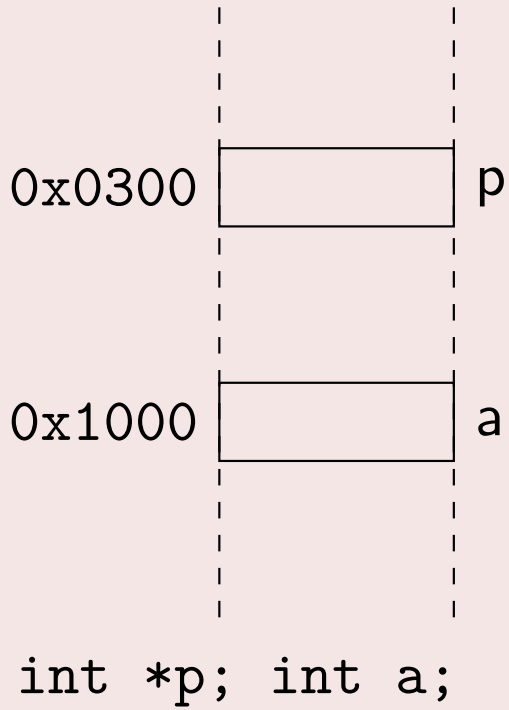
Représentation

Représentation mémoire

Exemple (Code source en C)

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

Représentation mémoire



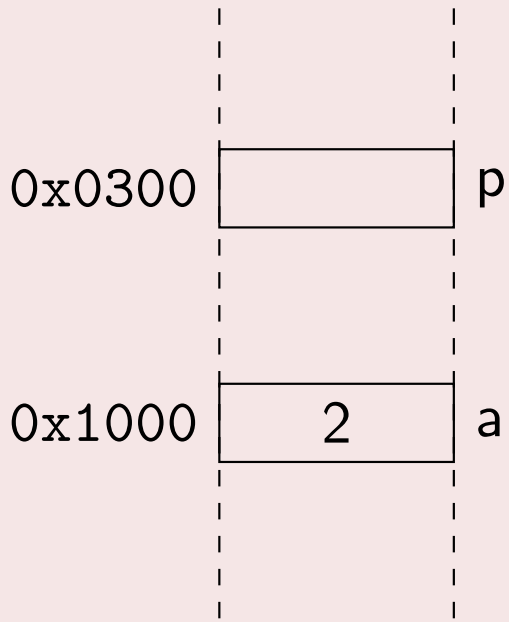
Représentation

Représentation mémoire

Exemple (Code source en C)

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

Représentation mémoire



a = 2;

Représentation

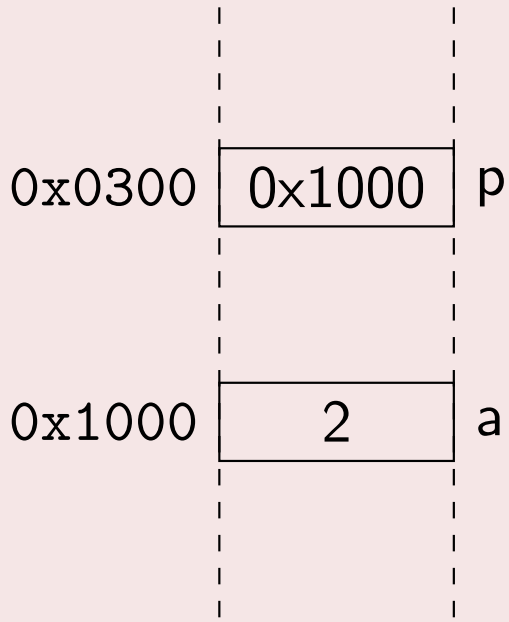
Représentation mémoire

★

Exemple (Code source en C)

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

Représentation mémoire



Plan

Définition et représentation ⋆⋆

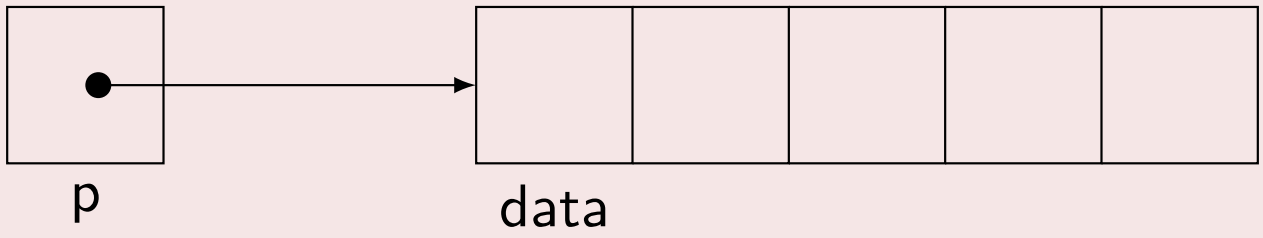
Définition (Tableau)

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 contiguë en mémoire.

Exemple (Code source en C)

```
int *p;
int data[5];
p = data; // p = &data[0];
```

Représentation générique

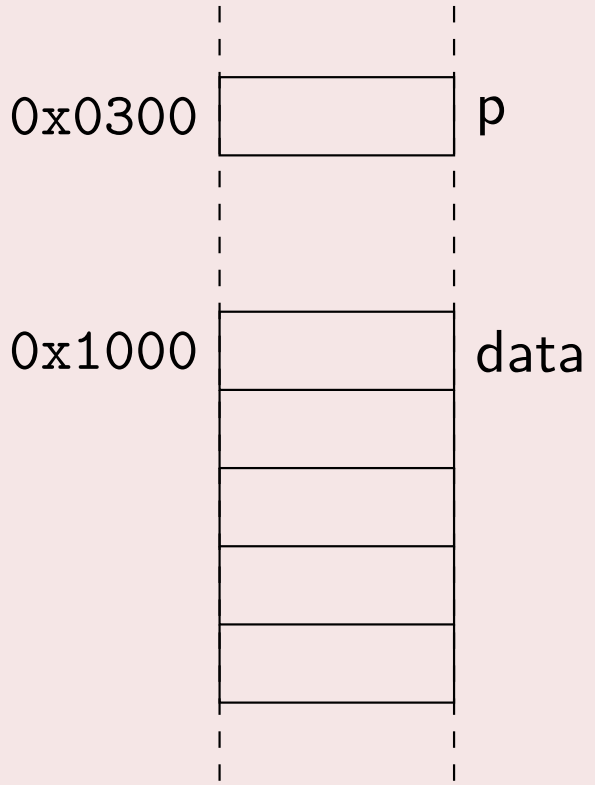


Représentation mémoire

Exemple (Code source en C)

```
int *p;
int data[5];
p = data; // p = &data[0];
```

Représentation mémoire

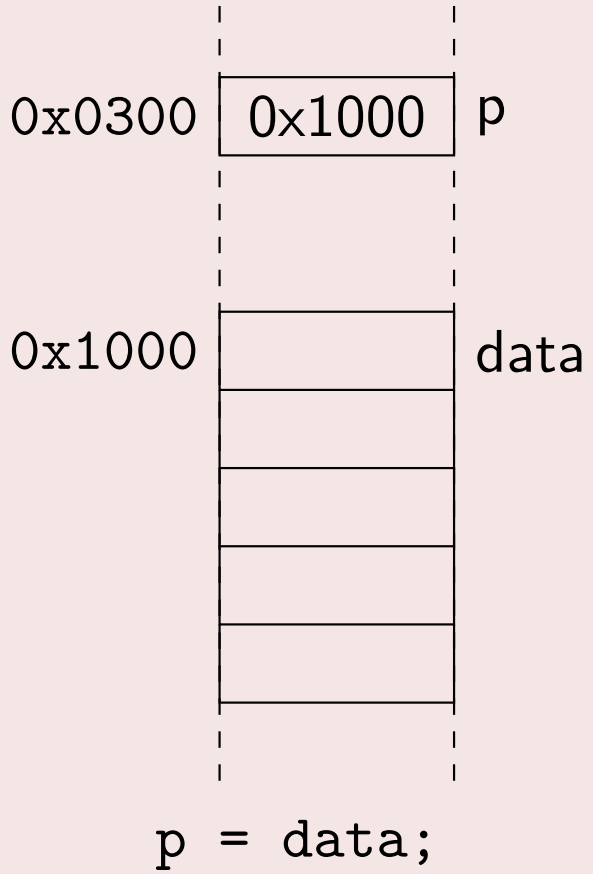


Représentation mémoire

Exemple (Code source en C)

```
int *p;
int data[5];
p = data; // p = &data[0];
```

Représentation mémoire



Plan

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 élémentaire sur les tableaux

- *accès aléatoire* à l'élément d'indice i : `data[i]`
  Complexité : $O(1)$

Recherche dans un tableau ⋆⋆

Définition du problème

Problème *Recherche d'un élément dans un tableau*

Données : un tableau *data* de taille *n* et un élément *e*

Résultat : l'indice de l'élément *e* dans le tableau *data* ou *n* si l'élément n'est pas dans le tableau

Algorithme

```
size_t array_search(const int *data, size_t n, int e) {
  size_t i = 0;
  while (i < n && data[i] != e) {
    i++;
  }
  return i;
}
```

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 *data* de taille *n* trié par ordre croissant et un élément *e*

Résultat : l'indice de l'élément *e* dans le tableau *data* 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 *lo* et *hi* qui sont les indices de début et de fin de la recherche, plus précisément l'indice de l'élément *e* se trouve dans l'intervalle $[lo; hi[$.

Recherche dans un tableau trié

★★★

Algorithme de recherche dichotomique

```
size_t array_binary_search(const int *data, size_t n,
                           int e, size_t lo, size_t hi) {
  if (lo == hi) {
    return n;
  }
  size_t mid = (lo + hi) / 2;
  if (e < data[mid]) {
    return array_binary_search(data, n, e, lo, mid);
  }
  if (data[mid] < e) {
    return array_binary_search(data, n, e, mid + 1, hi);
  }
  return mid;
}
```

Recherche dans un tableau trié

★★★

Algorithme général

```
size_t array_search_sorted(const int *data, size_t n,
                                                int e) {
  return array_binary_search(data, n, e, 0, n);
}
```

Complexité

L'opération fondamentale est la comparaison ($<$ et $=$). Pour un tableau de taille n (c'est-à-dire $hi - lo = n$), 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 ⋆⋆

Définition du problème

Problème *Insertion d'un élément dans un tableau*

Données : un tableau *data* de taille n occupé par $m < n$ éléments et un élément e à insérer à l'indice $j \in [0, m]$

Résultat : le tableau *data* 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

```
void array_insert(int *data, size_t m, int e, size_t j) {
  for (size_t i = m; i > j; --i) {
    data[i] = data[i - 1];
  }
  data[j] = e;
}
```

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 ⋆⋆

Définition du problème

Problème *Suppression d'un élément dans un tableau*

Données : un tableau *data* de taille n occupé par $m \leq n$ éléments et un élément à supprimer à l'indice $j \in [0, m[$

Résultat : le tableau *data* 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

```
void array_remove(int *data, size_t n, size_t j) {
  for (size_t i = j + 1; i < n; ++i) {
    data[i - 1] = data[i];
  }
}
```

Complexité

L'opération fondamentale est l'affectation ($\leftarrow$). 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 ⋆⋆

Définition du problème

Problème *Taille d'une chaîne de caractères*

Données : une chaîne de caractères *str*

Résultat : la taille de la chaîne *str*, 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

```
size_t str_length(const char *str) {
  size_t len = 0;
  while (str[len] != '\0') {
    len++;
  }
  return len;
}
```

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 ⋆⋆

Définition du problème

Problème *Comparaison de chaînes de caractères*

Données : deux chaînes de caractères str_1 et str_2

Résultat : un entier strictement négatif/nul/strictement positif suivant que la chaîne str_1 est inférieure/égale/supérieure à la chaîne str_2 selon l'ordre lexicographique

Remarque

La fonction C équivalente est `strcmp(3)`

Comparaison de chaînes de caractères ⋆⋆

Algorithme

```
int str_compare(const char *str1, const char *str2) {
  size_t i = 0;
  while (str1[i] != '\0' && str2[i] != '\0') {
    if (str1[i] != str2[i]) {
      return str1[i] - str2[i];
    }
    i++;
  }
  return str1[i] - str2[i];
}
```

Comparaison de chaînes de caractères ⋆⋆

Complexité

L'opération fondamentale est la comparaison ($\neq$). La complexité pour une chaîne str_1 de taille n_1 et une chaîne str_2 de taille n_2 est :

- Pire cas : les chaînes sont presque égales, sauf le dernier caractère (par exemple : «aaaa» et «aaab»), on fait $\min(n_1, n_2)$ comparaisons.

$$\mathcal{C}_{\text{worst}}(n_1, n_2) = \min(n_1, n_2) = O(\min(n_1, n_2))$$

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

$$\mathcal{C}_{\text{avg}}(n_1, n_2) = 1 = O(1)$$

Recherche d'une sous-chaîne ⋆⋆

Définition du problème

Problème *Recherche d'une sous-chaîne*

Données : une chaîne de caractères *haystack* dans laquelle on va chercher et une chaine de caractère *needle* que l'on va chercher

Résultat : un pointeur sur le début de la sous-chaîne *needle* dans la chaîne de caractères *haystack* ou *NULL* en cas d'échec

Remarque

La fonction C équivalente est `strstr(3)`

Recherche d'une sous-chaîne ⋆⋆

Algorithme

```
const char *str_search(const char *h, const char *n) {
  size_t i = 0;
  while (h[i] != '\0') {
    size_t j = 0;
    while (n[j] && h[i + j] && n[j] == h[i + j]) {
      j++;
    }
    if (n[j] == '\0') {
      return h + i;
    }
    i++;
  }
  return NULL;
}
```

Recherche d'une sous-chaîne

★★★

Complexité

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

- Pire cas : on échoue à la dernière lettre de *needle* 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 *needle* jusqu'à trouver la chaîne (en moyenne au milieu de la chaîne *haystack*), alors on fait $\frac{n}{2} + m$ comparaisons.

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

Plan

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 :

```
struct array {
  int *data;          // tableau
  size_t capacity;    // nombre d'éléments maximum du tableau
  size_t size;        // nombre d'éléments du tableaux
};
```

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 ⋆⋆⋆

Définition du problème

Problème *Insertion en fin d'un tableau dynamique*

Données : un tableau dynamique *self* et un élément *e* à insérer

Résultat : le tableau dynamique *self* 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

```
void array_add(struct array *self, int e) {
  if (self->size == self->capacity) {
    self->capacity += 1;
    int *data = calloc(self->capacity, sizeof(int));
    memcpy(data, self->data, self->size * sizeof(int));
    free(self->data);
    self->data = data;
  }
  self->data[self->size] = e;
  self->size += 1;
}
```

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 `memcpy` (qui est en $O(size)$), 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

```
void array_add(struct array *self, int e) {
  if (self->size == self->capacity) {
    self->capacity *= A;
    int *data = calloc(self->capacity, sizeof(int));
    memcpy(data, self->data, self->size * sizeof(int));
    free(self->data);
    self->data = data;
  }
  self->data[self->size] = e;
  self->size += 1;
}
```

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 : $1 + A + A^2 + A^3 + \ldots + A^k = \frac{1-A^{k+1}}{1-A} = O(A^k)$ copies d'éléments du tableaux et n ajout de e. Or, $k = \lfloor \log_A n \rfloor$, donc $O(A^k) = O(n)$, 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.

Quatrième partie

Listes chaînées

Plan de ce cours

Plan

Liste
Définition

★★★

Définition (Liste)

Une *liste* est :

- soit la liste vide, noté $\varnothing$
- soit un couple $\langle e, l \rangle$ où e est un élément et l est une liste

Remarque

Une liste est un objet mathématique récursif

Exemple (Listes)

- $\langle 2, \langle 3, \varnothing \rangle \rangle$ est une liste avec les éléments 2 et 3
- $\varnothing$ est une liste, donc $\langle 3, \varnothing \rangle$ est une liste, donc $\langle 2, \langle 3, \varnothing \rangle \rangle$ est une liste
- $\langle 2, 3 \rangle$ n'est pas une liste

Liste
Opérations élémentaires

★★

Opérations élémentaires

- *constructeur*, noté `cons`, définie par :

$$\texttt{cons}(e, l) = \langle e, l \rangle$$

- *test du vide*, noté `empty`, qui est définie par :

$$\texttt{empty}(l) = (l \stackrel{?}{=} \varnothing)$$

- *accès au premier élément*, noté `head`, définie par :

$$\texttt{head}(\langle e, l \rangle) = e$$

- *accès au reste de la liste*, noté `tail`, définie par :

$$\texttt{tail}(\langle e, l \rangle) = l$$

Liste chaînée
Définition

★★

Définition (Liste chaînée)

Une *liste chaînée* (*linked list*) est l'implémentation d'une liste. Elle est constituée de maillons ou nœuds (*node*) On distingue :

- les listes simplement chaînées, dans lesquelles chaque maillon est relié à son suivant
- les listes doublement chaînées, dans lesquelles chaque maillon est relié à son suivant et à son précédent

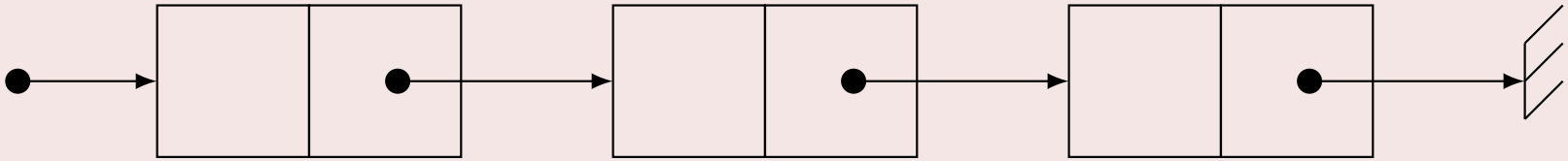
Remarque

La plupart des algorithmes sont valables pour les deux types de listes. Certains algorithmes sont plus efficaces avec des listes doublement chaînées, au prix d'une gestion de la liste plus difficile de manière générale.

Liste chaînée

Représentation et implémentation

Représentation



Implémentation

```
struct list {
  int data;
  struct list *next;
};
```

Plan

Taille d'une liste

★★

Définition du problème

Problème *Taille d'une liste*

Données : une liste *l*

Résultat : la taille de la liste, c'est-à-dire le nombre d'éléments contenus dans la liste

Algorithme

```
size_t list_size(const struct list *self) {
  if (self == NULL) {
    return 0;
  }
  return 1 + list_size(self->next);
}
```

Taille d'une liste

★★

Complexité

L'opération fondamentale est l'addition $(+)$. La complexité pour une liste de taille n 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 :

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

Accès au j^e élément d'une liste ⋆⋆

Définition du problème

Problème *Accès au j^e élément d'une liste*

Données : une liste l de taille $n \geq 1$ et un indice j, avec $j < n$

Résultat : un pointeur sur l'élément situé à l'indice j dans la liste

Algorithme

```
int *list_access(const struct list *self, size_t j) {
  if (self == NULL) {
    return NULL;
  }
  if (j == 0) {
    return &self->data;
  }
  return list_access(self->next, j - 1);
}
```

Accès au j^e élément d'une liste ⋆⋆

Complexité

L'opération fondamentale est l'appel à `->next`. La complexité de cet algorithme est de j appels. Donc, en moyenne, si on accède à l'élément d'indice $\frac{n}{2}$, la complexité est :

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

Recherche d'un élément dans une liste ⋆⋆

Définition du problème

Problème *Recherche d'un élément dans une liste*

Données : une liste l de taille n et un élément e

Résultat : l'indice de l'élément e dans la liste l ou n si l'élément n'est pas dans la liste

Variantes

Il existe plusieurs variantes de cette fonction, suivant ce qu'on souhaite faire de l'élément recherché :

- on peut renvoyer un booléen qui indique si l'élément e est dans la liste l
- on peut renvoyer la sous-liste dont le premier élément est l'élément e recherché, ou $\varnothing$ si l'élément n'est pas dans la liste l

Recherche d'un élément dans une liste ⋆⋆

Algorithme

```
size_t list_search(const struct list *self, int e) {
  if (self == NULL) {
    return 0;
  }
  if (self->data == e) {
    return 0;
  }
  return 1 + list_search(self->next, e);
}
```

Recherche d'un élément dans une liste ⋆⋆

Complexité

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

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

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

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

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

Insertion d'un élément après l'élément courant ⋆⋆

Définition du problème

Problème *Insertion d'un élément après l'élément courant*

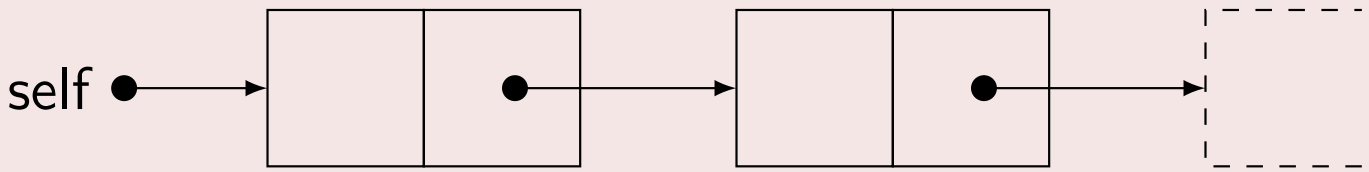
Données : une liste l de taille $n \geq 1$ et un élément e

Résultat : la liste l de taille $n + 1$ avec l'élément e placé après l'élément courant

Insertion d'un élément après l'élément courant ⋆⋆

Algorithme

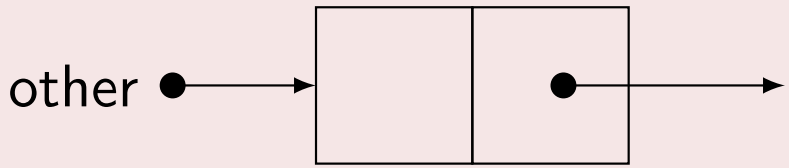
```
void list_insert_after(struct list *self, int e) {
  struct list *other = malloc(sizeof(struct list));
  other->data = e;
  other->next = self->next;
  self->next = other;
}
```

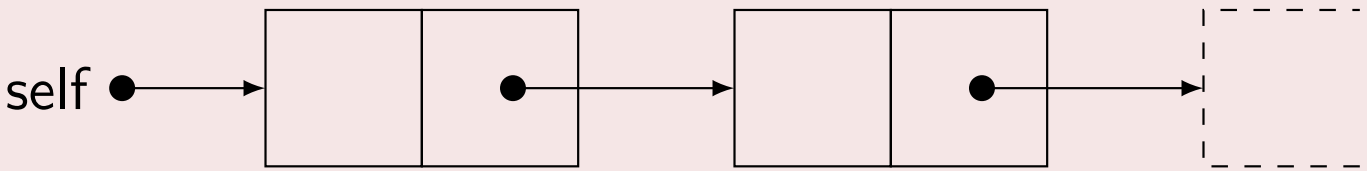


Insertion d'un élément après l'élément courant ⋆⋆

Algorithme

```
void list_insert_after(struct list *self, int e) {
  struct list *other = malloc(sizeof(struct list));
  other->data = e;
  other->next = self->next;
  self->next = other;
}
```

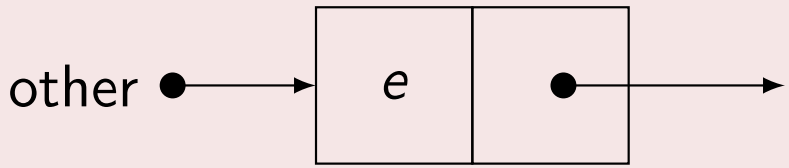


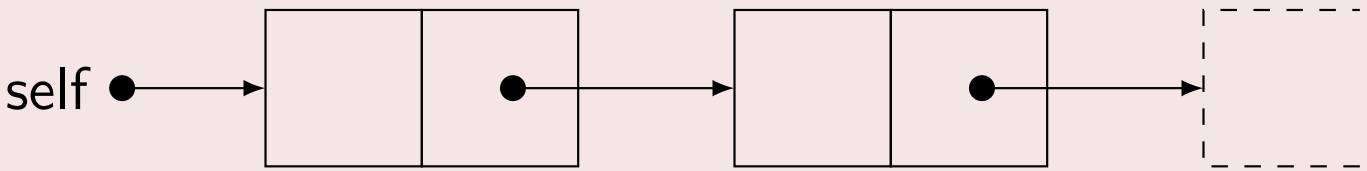


Insertion d'un élément après l'élément courant ⋆⋆

Algorithme

```
void list_insert_after(struct list *self, int e) {
  struct list *other = malloc(sizeof(struct list));
  other->data = e;
  other->next = self->next;
  self->next = other;
}
```

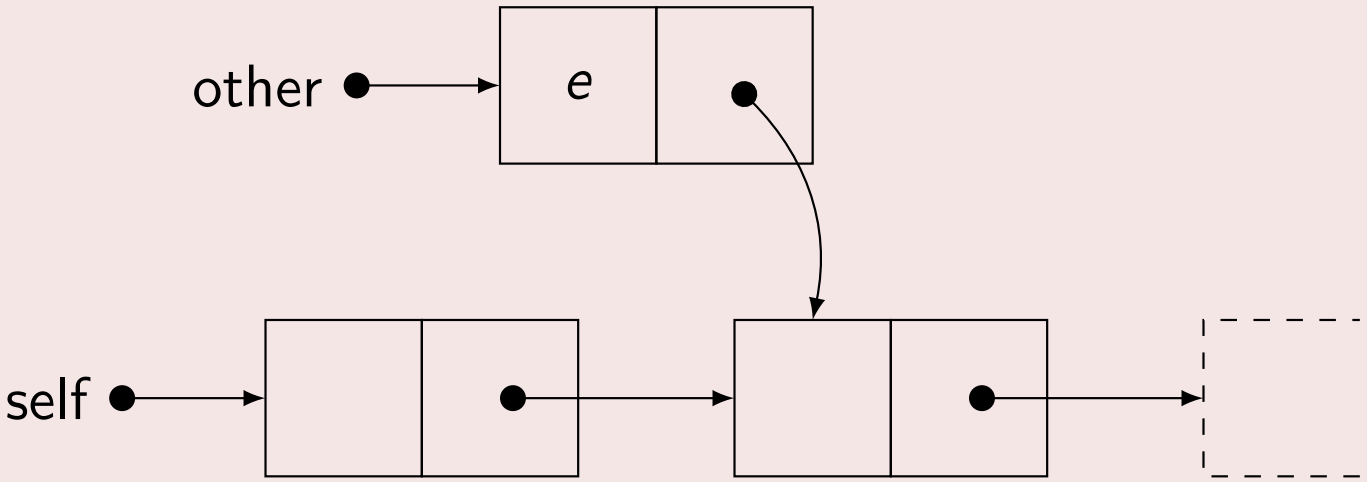




Insertion d'un élément après l'élément courant ⋆⋆

Algorithme

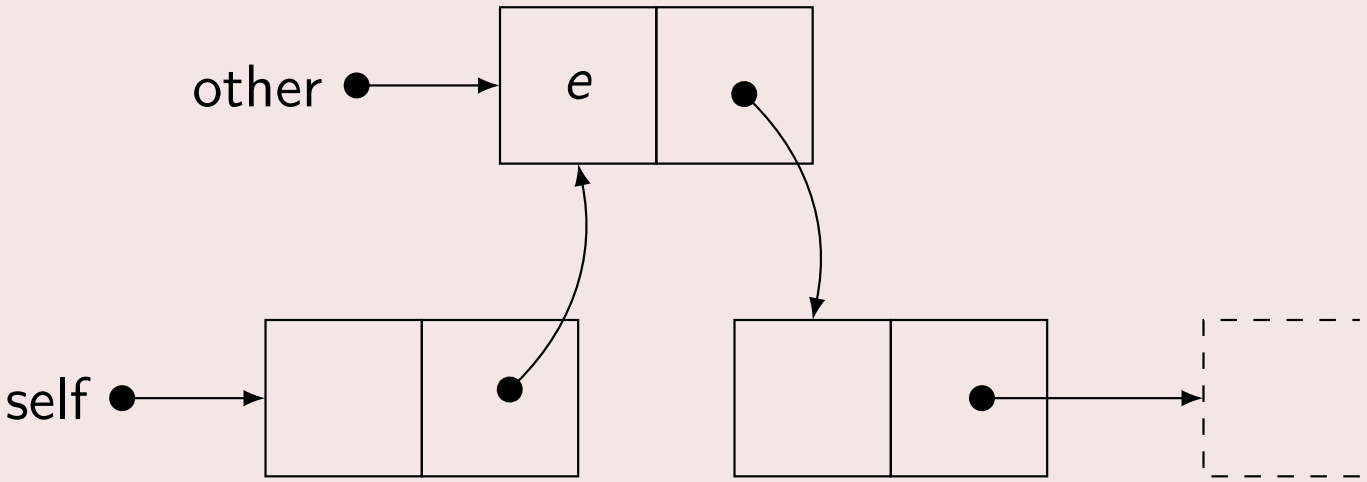
```
void list_insert_after(struct list *self, int e) {
  struct list *other = malloc(sizeof(struct list));
  other->data = e;
  other->next = self->next;
  self->next = other;
}
```



Insertion d'un élément après l'élément courant ⋆⋆

Algorithme

```
void list_insert_after(struct list *self, int e) {
  struct list *other = malloc(sizeof(struct list));
  other->data = e;
  other->next = self->next;
  self->next = other;
}
```



Suppression d'un élément après l'élément courant ⋆⋆

Définition du problème

Problème *Suppression d'un élément après l'élément courant*

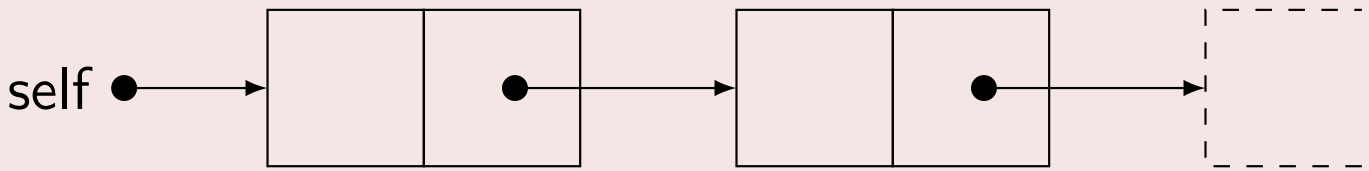
Données : une liste l de taille $n \geq 2$

Résultat : la liste l avec $n - 1$ éléments

Suppression d'un élément après l'élément courant ⋆⋆

Algorithme

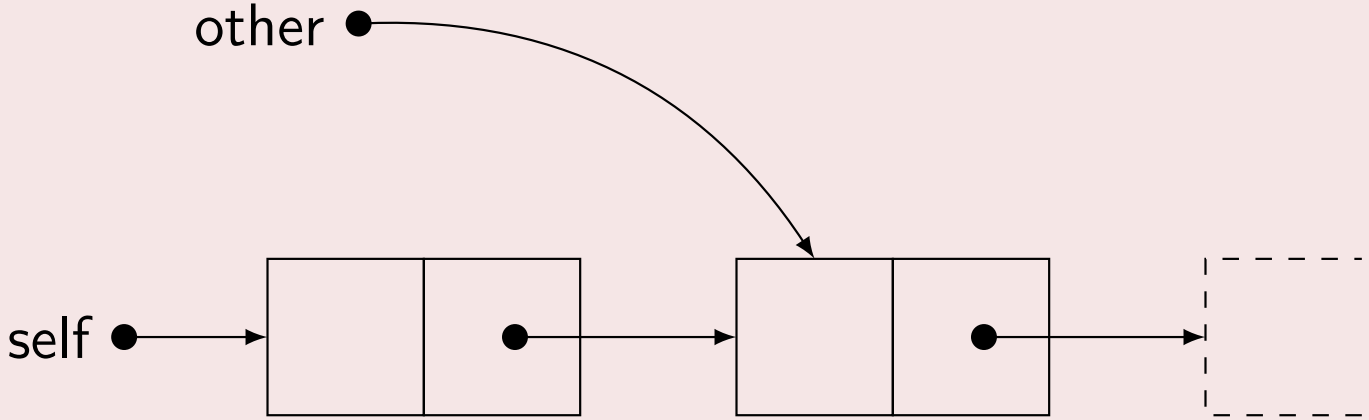
```
void list_remove_after(struct list *self) {
  struct list *other = self->next;
  self->next = other->next;
  free(other);
}
```



Suppression d'un élément après l'élément courant ⋆⋆

Algorithme

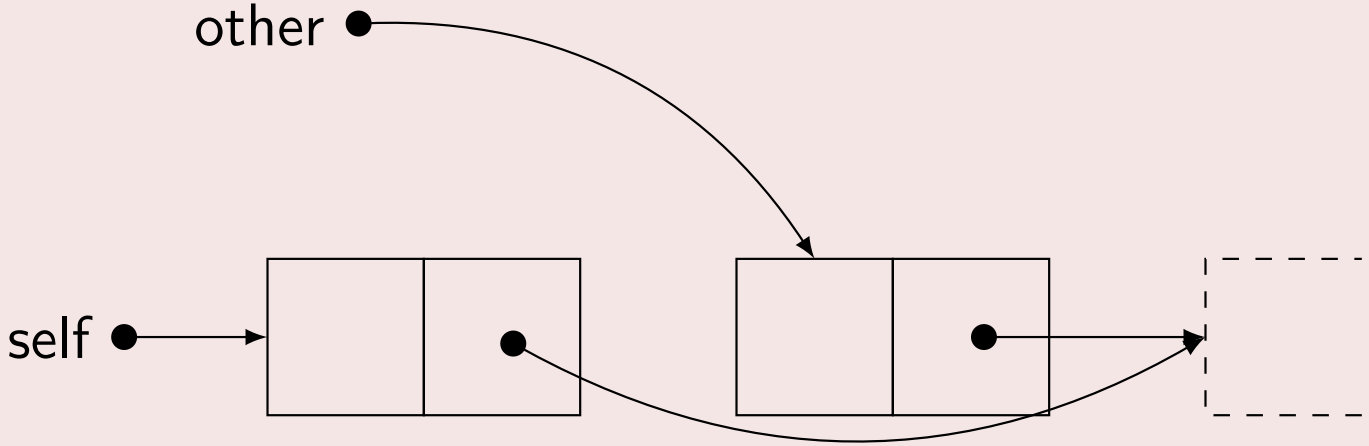
```
void list_remove_after(struct list *self) {
  struct list *other = self->next;
  self->next = other->next;
  free(other);
}
```



Suppression d'un élément après l'élément courant ⋆⋆

Algorithme

```
void list_remove_after(struct list *self) {
  struct list *other = self->next;
  self->next = other->next;
  free(other);
}
```



Suppression d'un élément après l'élément courant ⋆⋆

Algorithme

```
void list_remove_after(struct list *self) {
  struct list *other = self->next;
  self->next = other->next;
  free(other);
}
```

Insertion d'un élément en fin de liste ⋆⋆

Définition du problème

Problème *Insertion d'un élément en fin de liste*

Données : une liste l de taille n et un élément e

Résultat : la liste l de taille $n+1$ avec l'élément e en dernière position

Remarque

Pour une liste doublement chaînée, cette opération est en temps constant. Nous allons voir un algorithme pour les listes simplement chaînées.

Insertion d'un élément en fin de liste ⋆⋆

Algorithme

```
struct list *list_insert_back(struct list *self, int e) {
  if (self == NULL) {
    struct list *last = malloc(sizeof(struct list));
    last->data = e;
    last->next = NULL;
    return last;
  }
  self->next = list_insert_back(self->next, e);
  return self;
}
```

Insertion d'un élément en fin de liste ⋆⋆

Complexité

L'opération fondamentale est l'appel à `->next`. La complexité pour une liste l de taille n est :

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

Suppression d'un élément en fin de liste ⋆⋆

Définition du problème

Problème *Suppression d'un élément en fin de liste*

Données : une liste l de taille $n \geq 1$

Résultat : la liste l avec $n - 1$ éléments

Remarque

Pour une liste doublement chaînée, cette opération est en temps constant.
Nous allons voir un algorithme pour les listes simplement chaînées.

Suppression d'un élément en fin de liste ⋆⋆

Algorithme

```
struct list *list_remove_back(struct list *self) {
  if (self->next == NULL) {
    free(self);
    return NULL;
  }
  self->next = list_remove_back(self->next);
  return self;
}
```

Suppression d'un élément en fin de liste ⋆⋆

Complexité

L'opération fondamentale est l'appel à `->next`. La complexité pour une liste l de taille n est :

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

Plan

Résumé des complexités

★★★

Résumé des complexités

Opération	Tableaux dynamiques	Listes simplement chaînées	Listes doublement chaînées
Accès au i^{e} élément	$O(1)$	$O(n)$	$O(n)$
Accès au premier élément	$O(1)$	$O(1)$	$O(1)$
Accès au dernier élément	$O(1)$	$O(n)$	$O(1)$
Recherche	$O(n)$	$O(n)$	$O(n)$
Recherche dans trié	$O(\log n)$	$O(n)$	$O(n)$
Insertion en début	$O(n)$	$O(1)$	$O(1)$
Insertion en fin	$O(1)$	$O(n)$	$O(1)$
Suppression en début	$O(n)$	$O(1)$	$O(1)$
Suppression en fin	$O(1)$	$O(n)$	$O(1)$

Plan

Pile
Définition

★★★

Définition (Pile)

Une *pile* (*stack*) est une structure de données dans laquelle on peut stocker des éléments en mode LIFO (*Last In, First Out*), c'est-à-dire «dernier arrivé, premier sorti». On peut comparer le fonctionnement à une pile d'assiette.

Pile

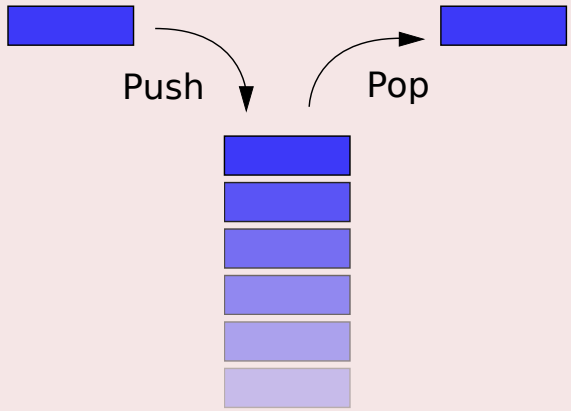
Opérations élémentaires

★★

Opérations élémentaires

- ajout d'un élément sur la pile, noté `push`
- retrait de l'élément sur la pile, noté `pop`
- accès à l'élément sur la pile, noté `peek`
- test du vide de la pile, noté `empty`

LIFO



Pile
Implémentations

★★

Implémentations d'une pile

On peut implémenter une pile de deux manières :

- en utilisant un tableau dynamique
- en utilisant une liste (simplement chaînée)

Opération	Tableau	Liste
`push`	$O(1)$	$O(1)$
`pop`	$O(1)$	$O(1)$
`peek`	$O(1)$	$O(1)$
`empty`	$O(1)$	$O(1)$

Plan

File
Définition

★★★

Définition (File)

Une *file* (*queue*) est une structure de données dans laquelle on peut stocker des éléments en mode FIFO (*First In, First Out*), c'est-à-dire «premier arrivé, premier sorti». On peut comparer le fonctionnement à une file d'attente.

Variantes

- File bornée : file avec un nombre d'élément maximum

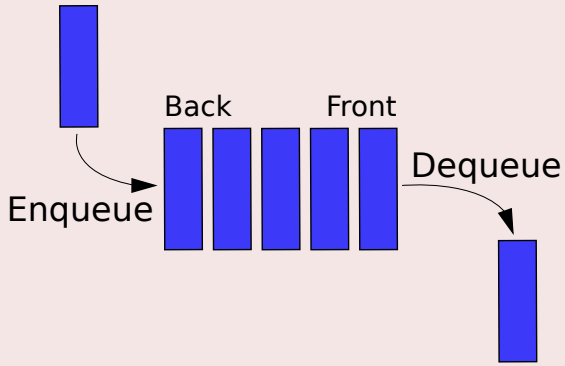
File
Opérations élémentaire

★★

Opérations élémentaires

- ajout d'un élément en fin de file, noté `enqueue`
- retrait de l'élément en début de file, noté `dequeue`
- accès à l'élément en début de file, noté `peek`
- test du vide de la file, noté `empty`

FIFO



File
Implémentation

★★

Implémentations d'une file

On peut implémenter une file de deux manières

- une liste doublement chaînée
- un tableau dynamique modifié pour fonctionner comme un tampon circulaire (*circular buffer*)

Opération	Liste	Tableau
`enqueue`	$O(1)$	$O(1)$
`dequeue`	$O(1)$	$O(1)$
`peek`	$O(1)$	$O(1)$
`empty`	$O(1)$	$O(1)$

Plan

File à double entrée

Définition

★★★

Définition (File à double entrée)

Une *file à double entrée* (*double-ended queue* ou *deque*) est une structure de données dans laquelle on peut ajouter ou supprimer des éléments à la fois au début et à la fin.

Remarque

Une file à double entrée est une généralisation d'une pile et d'une file.

File à double entrée

Opérations élémentaires

★★

Opérations élémentaires

- ajout d'un élément en début de file, noté `push`
- ajout d'un élément en fin de file, noté `inject`
- retrait d'un élément en début de file, noté `pop`
- retrait d'un élément en fin de file, noté `eject`
- accès à l'élément en début de file, noté `front`
- accès à l'élément en fin de file, noté `back`
- test du vide de la file, noté `empty`

File à double entrée
Implémentations

Implémentations

On peut implémenter une file à double entrée de deux manières :

- une liste doublement chaînée
- un tableau dynamique modifié pour fonctionner comme un tampon circulaire (*circular buffer*)

Cinquième partie

Algorithmes de tri (1/2)

Plan de ce cours

Plan

Problème du tri par comparaisons ★★★

Problème du tri par comparaisons

Problème *Tri par comparaisons d'un tableau*

Données : Un tableau t de taille n

Résultat : le tableau t trié par ordre croissant de ses éléments

Remarque

Les éléments du tableau peuvent être de n'importe quel type, pourvu qu'on puisse les comparer les uns avec les autres. La comparaison de ces éléments peut éventuellement être une opération coûteuse (exemple : chaîne de caractères).

Problème du tri par comparaisons ⋆⋆⋆

Opérations fondamentales

Les deux opérations fondamentales que l'on va considérer ici sont :

- la comparaison
- l'affectation (ou l'échange)

Algorithme d'échange

```
void array_swap(int *data, size_t i, size_t j) {
  int tmp = data[i];
  data[i] = data[j];
  data[j] = tmp;
}
```

Borne inférieure de la complexité ⋆⋆

Théorème (Borne inférieure de la complexité)

La complexité algorithmique en moyenne et en pire cas d'un algorithme de tri par comparaisons est au moins $O(n \log n)$. Les tris qui demandent $O(n \log n)$ opérations en moyenne sont dits optimaux.

Remarque

Il existe des algorithmes optimaux de complexité $O(n \log n)$ en moyenne.

Propriétés des tris

Tri stable

★★

Définition (Tri stable)

On dit qu'un tri est *stable* s'il conserve l'ordre relatif des éléments égaux.

Exemple

Pour l'ensemble de paires suivantes qu'on trie par le premier élément de la paire :

$$[(4,1);(3,2);(3,3);(5,4)]$$

un algorithme de tri stable donnera toujours le résultat suivant :

$$[(3,2);(3,3);(4,1);(5,4)]$$

et jamais le résultat suivant :

$$[(3,3);(3,2);(4,1);(5,4)]$$

Propriétés des tris

Tri en place

★★

Définition (Tri en place)

On dit qu'un tri est *en place* ou *sur place* s'il modifie directement la structure de données à trier. Autrement dit, pour n éléments, la complexité en mémoire est en $O(1)$.

Cas d'un tableau déjà trié

Problématique

★★

Cas d'un tableau déjà trié

Si un tableau est déjà trié, alors il n'est pas nécessaire d'appliquer un algorithme pour le trier.

- Comment faire pour savoir si un tableau est déjà trié ?
- Est-ce efficace de vérifier qu'un tableau est déjà trié ?

On s'intéresse donc au problème suivant :

Problème *Tabeau trié*

Données : Un tableau t de taille n

Question : Le tableau t est-il trié par ordre croissant ?

Cas d'un tableau déjà trié

Algorithme

★★

Algorithme

```
bool array_is_sorted(int *data, size_t n) {
  for (size_t i = 1; i < n; ++i) {
    if (data[i - 1] > data[i]) {
      return false;
    }
  }
  return true;
}
```

Cas d'un tableau déjà trié

Complexité

★★

Complexité

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

- Pire cas : quand le tableau est trié, on fait $n-1$ comparaisons.

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

- En moyenne : le tableau n'est pas trié et on va échouer dès les premiers éléments.

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

Cas d'un tableau déjà trié

Conclusion

★★

Conclusion

- Si un tableau est déjà trié, appeler `array_is_sorted` va coûter $O(n)$ mais alors, on a fini.
- Si un tableau n'est pas trié, appeler `array_is_sorted` va coûter en moyenne $O(1)$ (et au pire $O(n)$) et on va le trier en $O(n \log n)$ au mieux donc la complexité globale sera au mieux de $O(n \log n)$

$\rightarrow$ Donc, on peut toujours vérifier qu'un tableau est déjà trié avant de le trier sans pénalité sur la complexité globale (et avec un gain dans le cas où il est effectivement trié).

Cas des listes chaînées ⋆⋆

Cas des listes chaînées

Certains algorithmes s'adaptent bien aux listes chaînées et ont la même complexité. D'autres algorithmes s'adaptent mais ont une complexité supérieure, ou ne s'adapte pas du tout aux listes chaînées.

Plan

Tri par sélection

Principe

★★★

Principe

Le *tri par sélection* consiste à rechercher le plus petit élément du tableau et à le placer en première position puis de réitérer avec le sous-tableau commençant à la deuxième position. Et ainsi de suite.

Tri par sélection

Algorithme

★★

Algorithme

```
void array_selection_sort(int *data, size_t n) {
  for (size_t i = 0; i < n - 1; ++i) {
    size_t j = i;
    for (size_t k = j + 1; k < n; ++k) {
      if (data[k] < data[j]) {
        j = k;
      }
    }
    array_swap(data, i, j);
  }
}
```

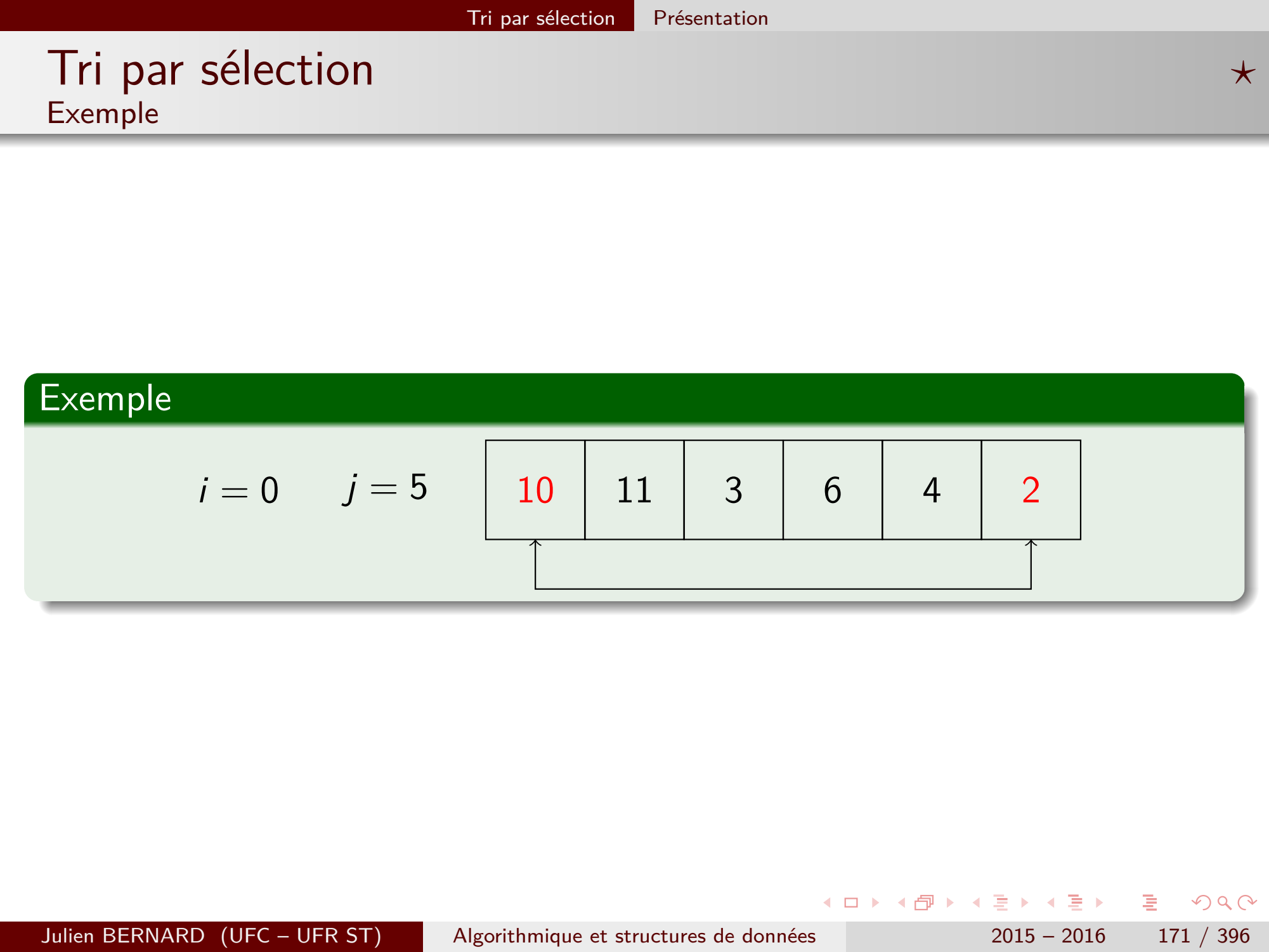
Tri par sélection
Exemple

Exemple

$i = 0$

10	11	3	6	4	2

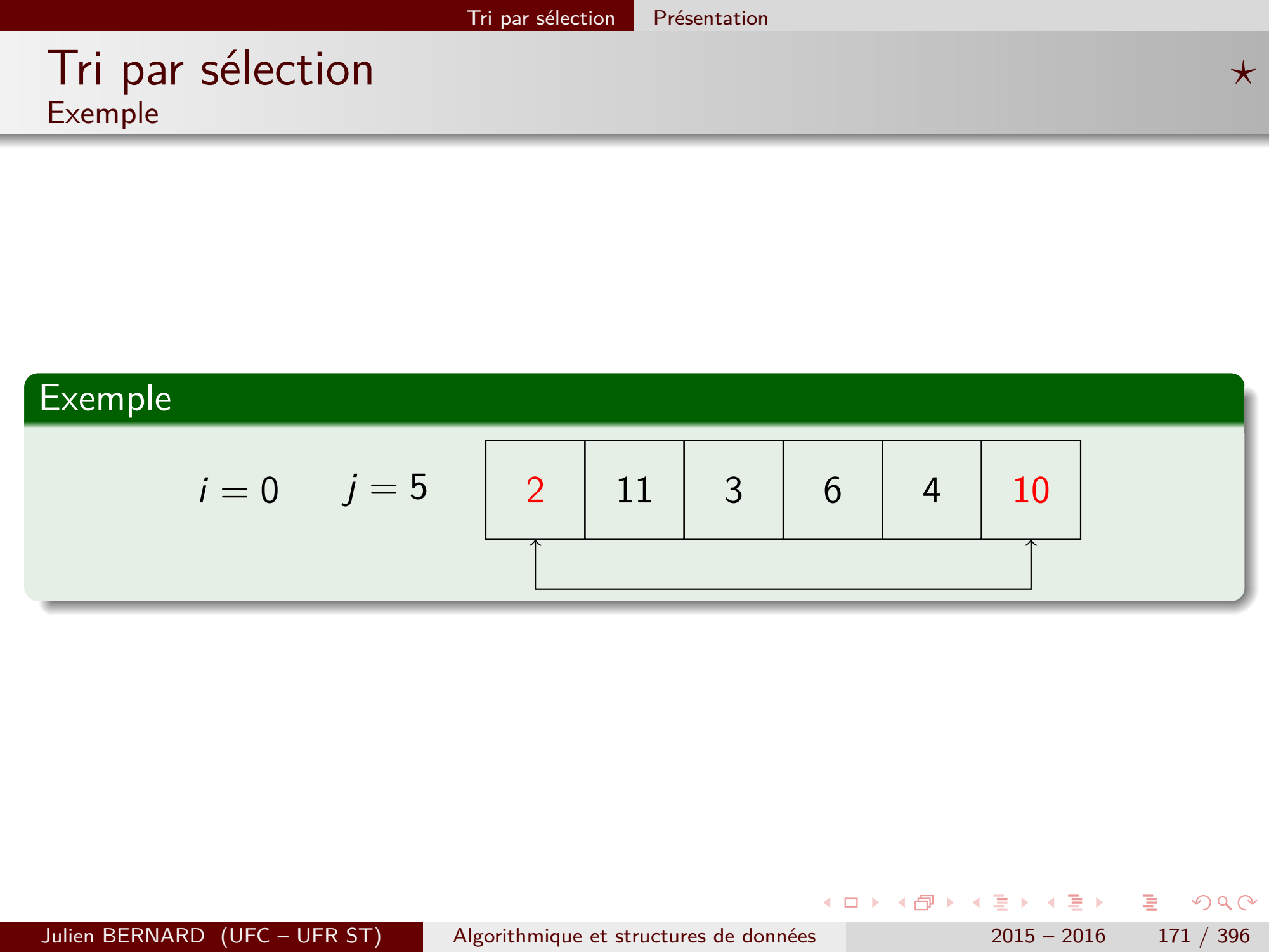
Tri par sélection
Exemple



Exemple

$i = 0$ $j = 5$

Tri par sélection
Exemple



Exemple

$i = 0$ $j = 5$

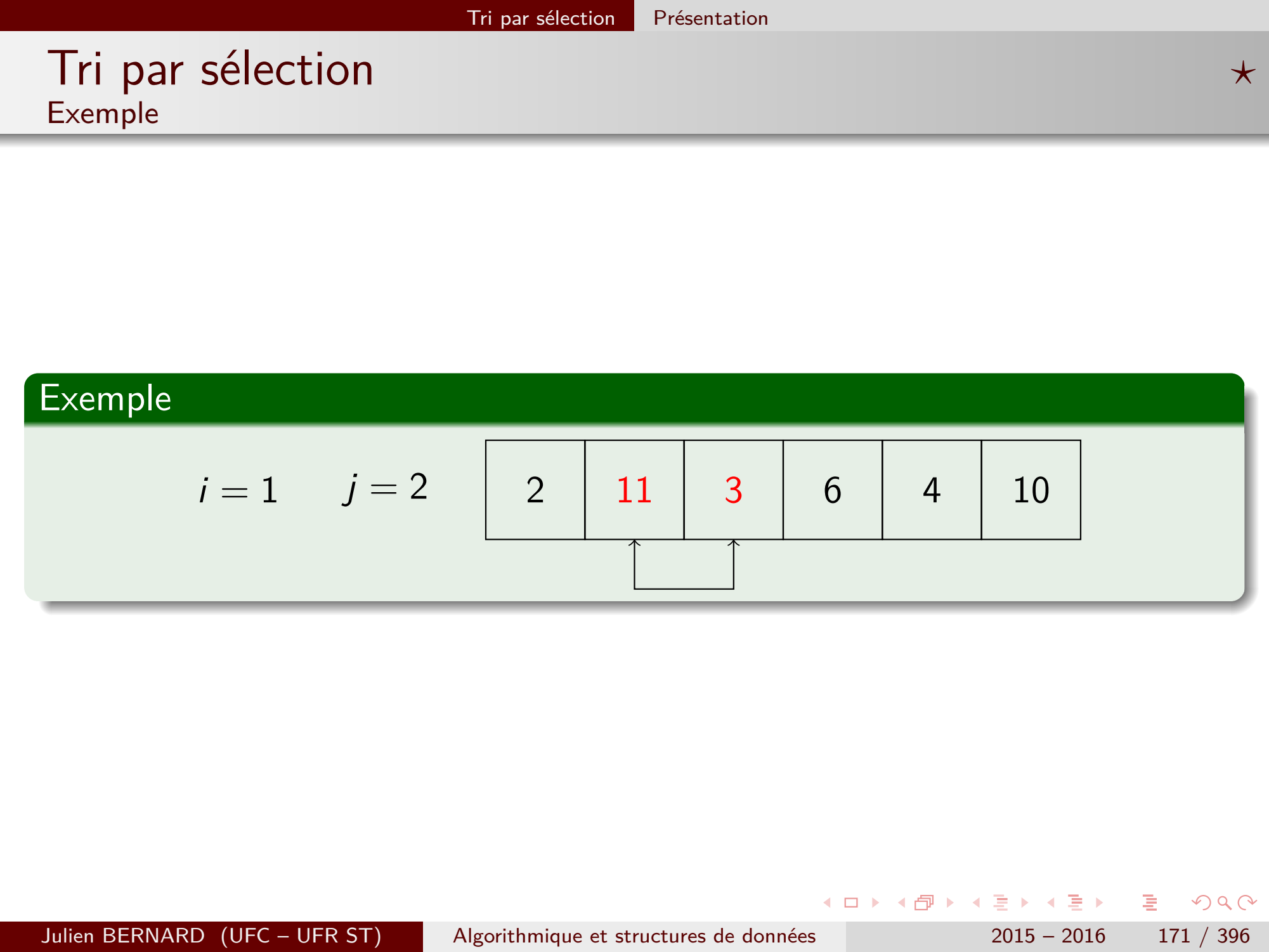
Tri par sélection
Exemple

Exemple

$i = 1$

2	11	3	6	4	10

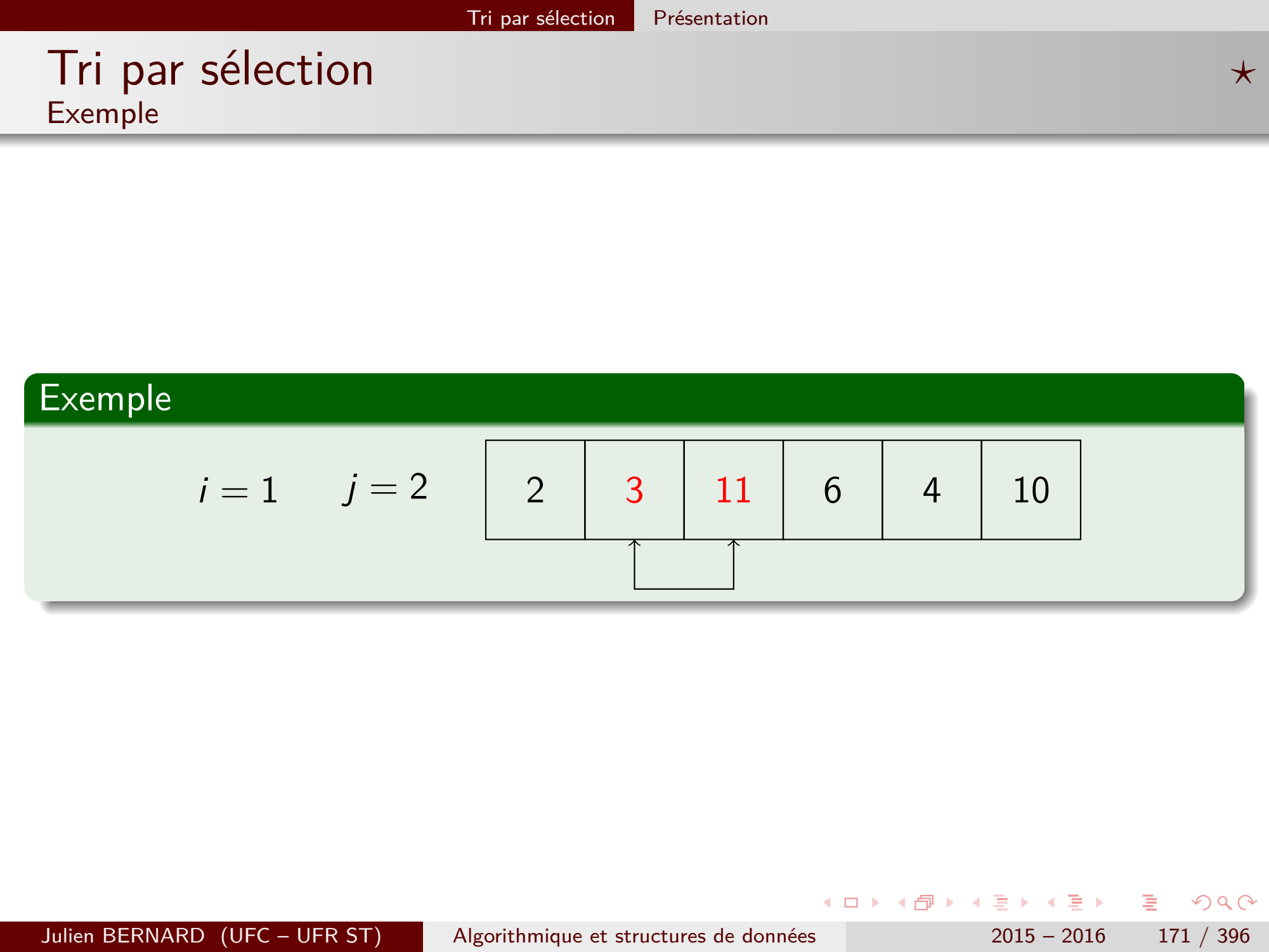
Tri par sélection
Exemple



Exemple

$i = 1 \quad j = 2$

Tri par sélection
Exemple



Exemple

$i = 1 \quad j = 2$

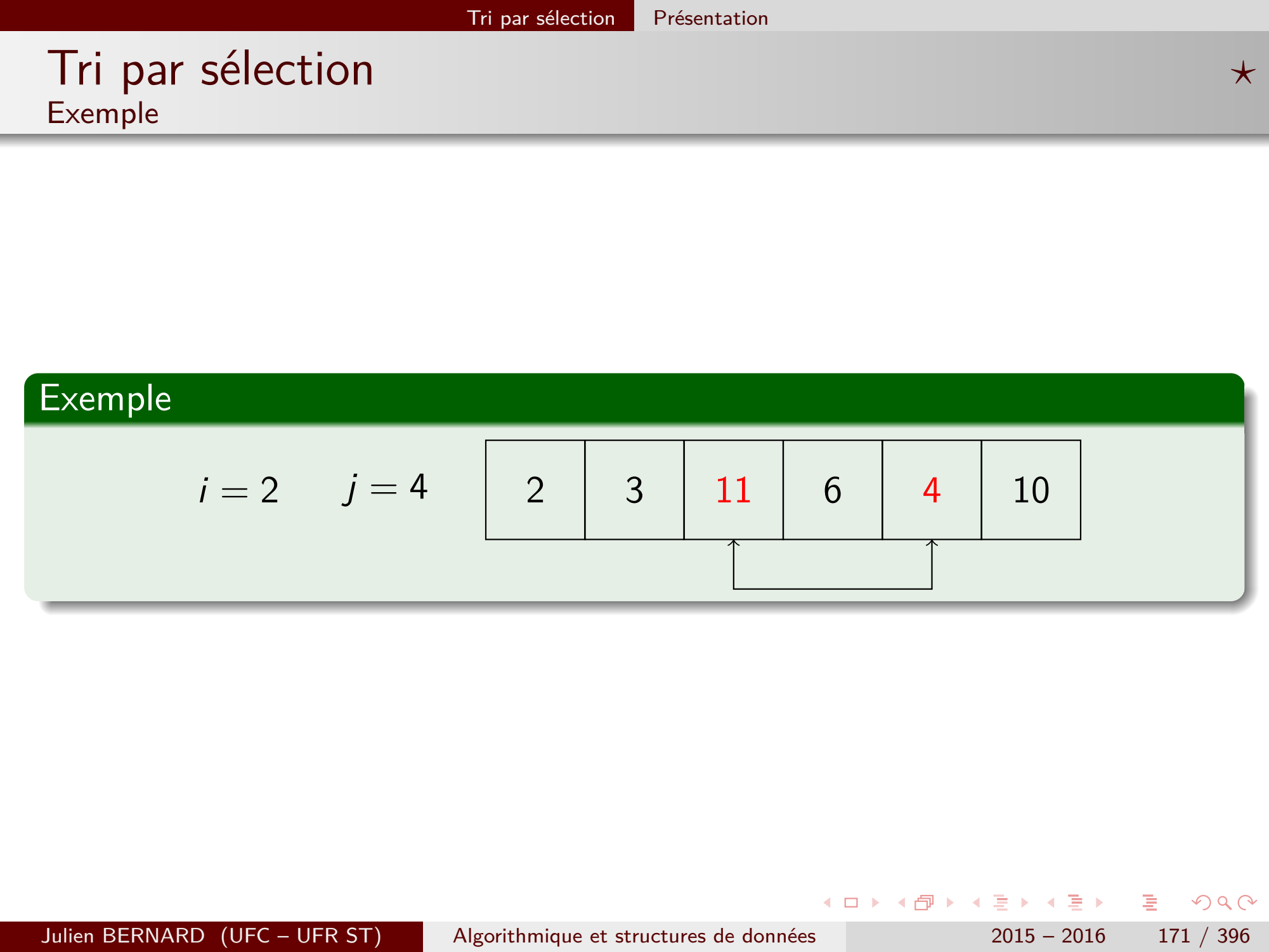
Tri par sélection
Exemple

Exemple

$i = 2$

2	3	11	6	4	10

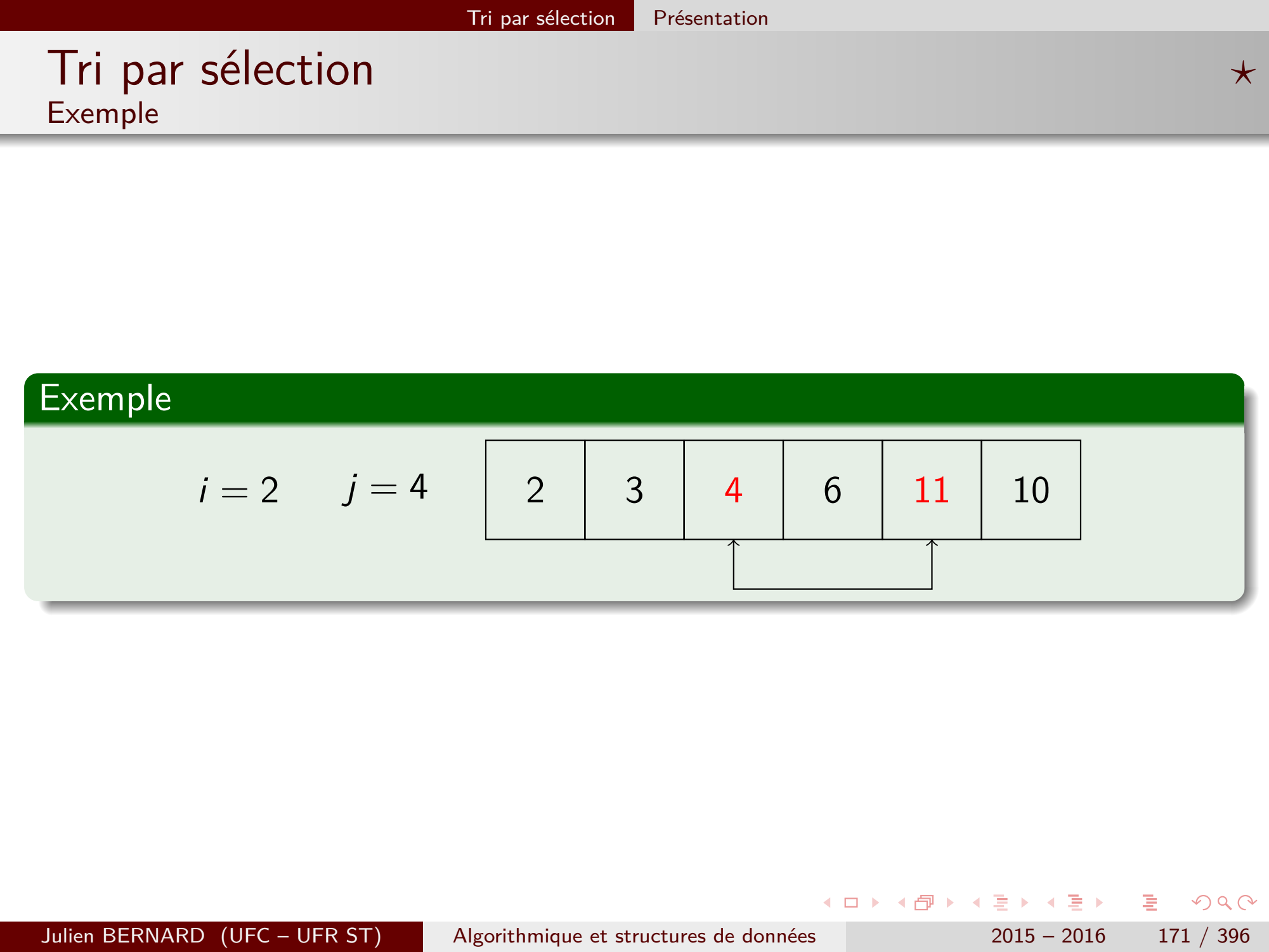
Tri par sélection
Exemple



Exemple

$i = 2 \qquad j = 4$

Tri par sélection
Exemple



Exemple

$i = 2 \qquad j = 4$

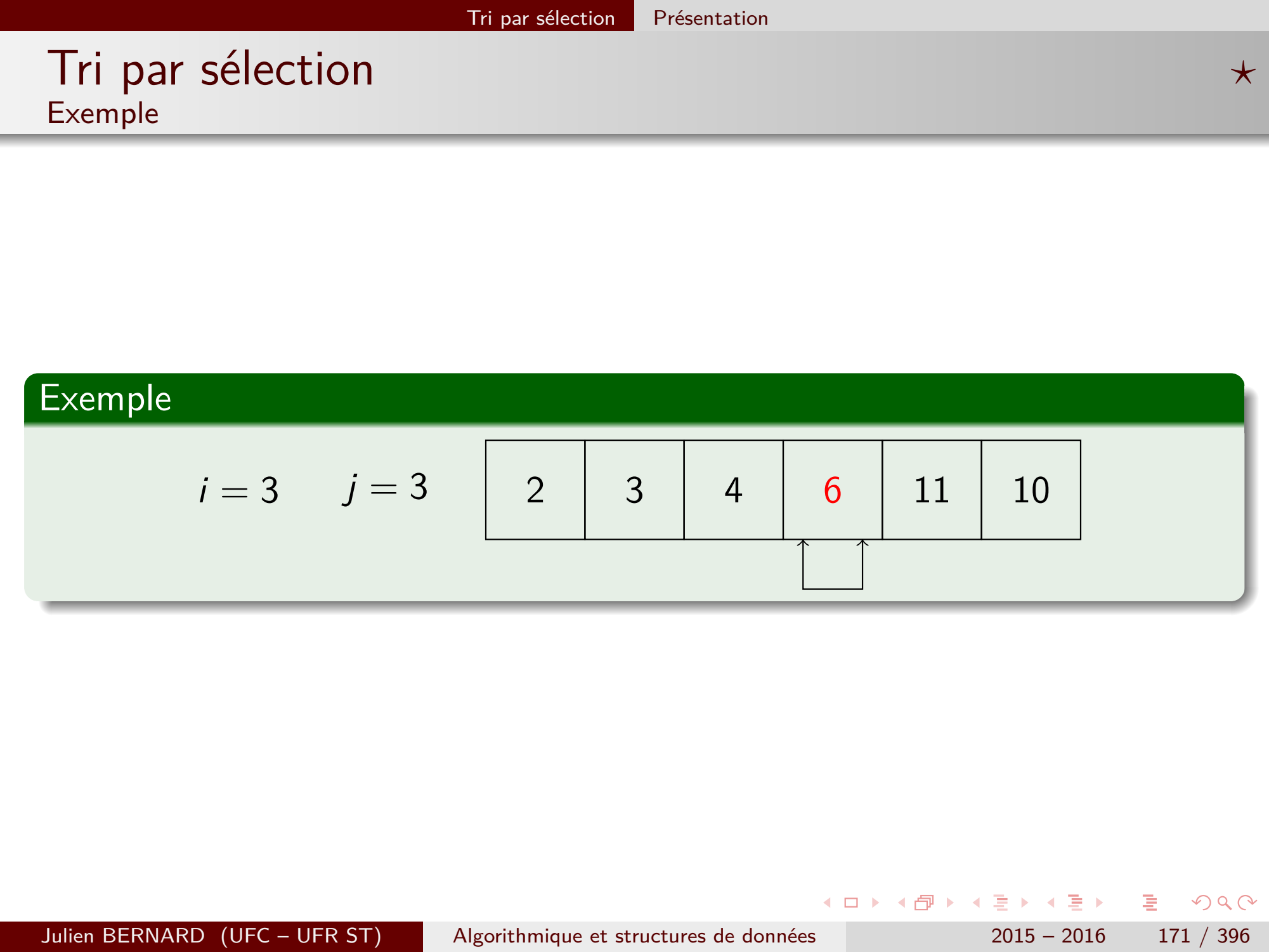
Tri par sélection
Exemple

Exemple

$i = 3$

2	3	4	6	11	10

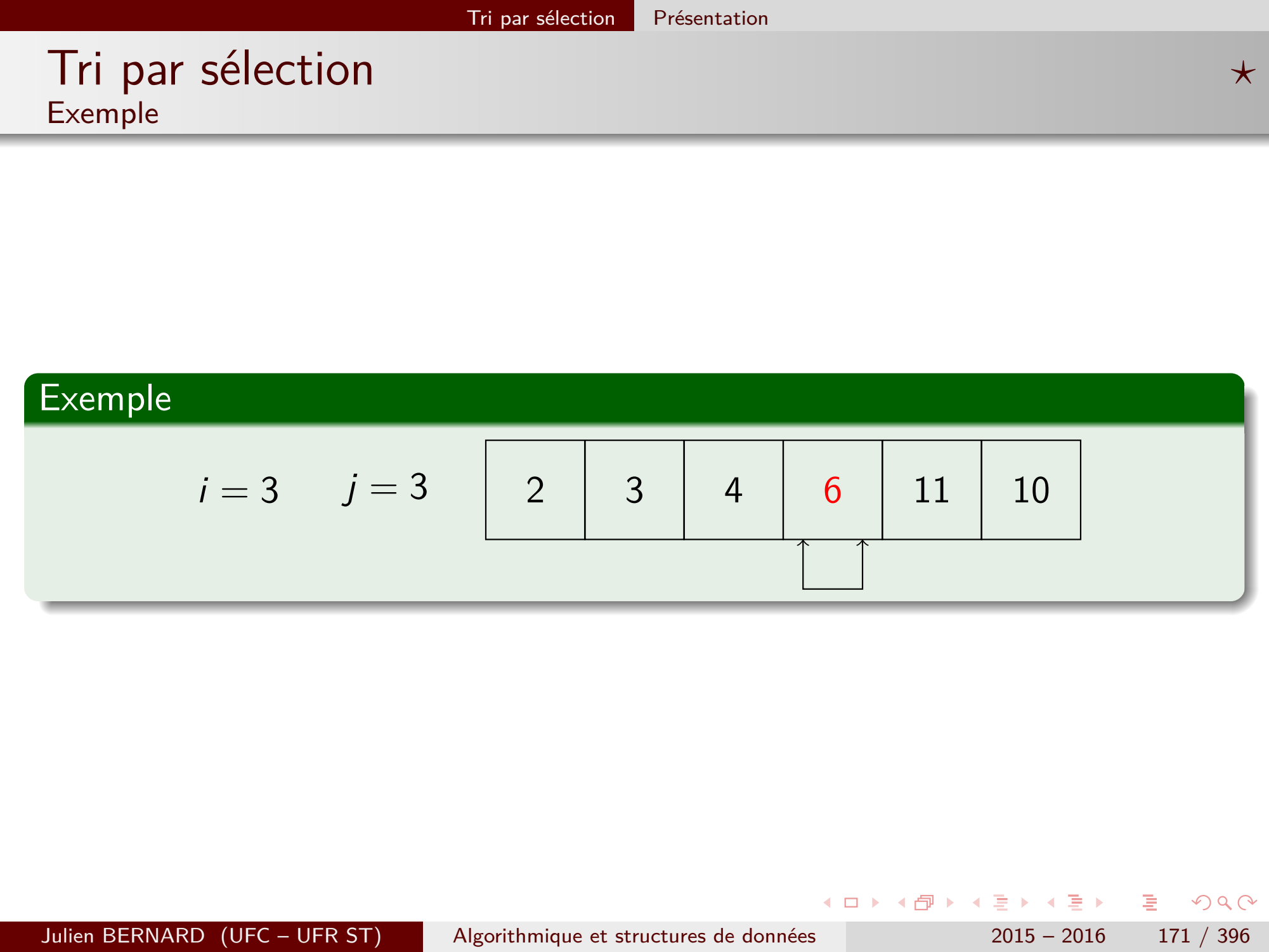
Tri par sélection
Exemple



Exemple

$i = 3 \quad j = 3$

Tri par sélection
Exemple



Exemple

$i = 3 \quad j = 3$

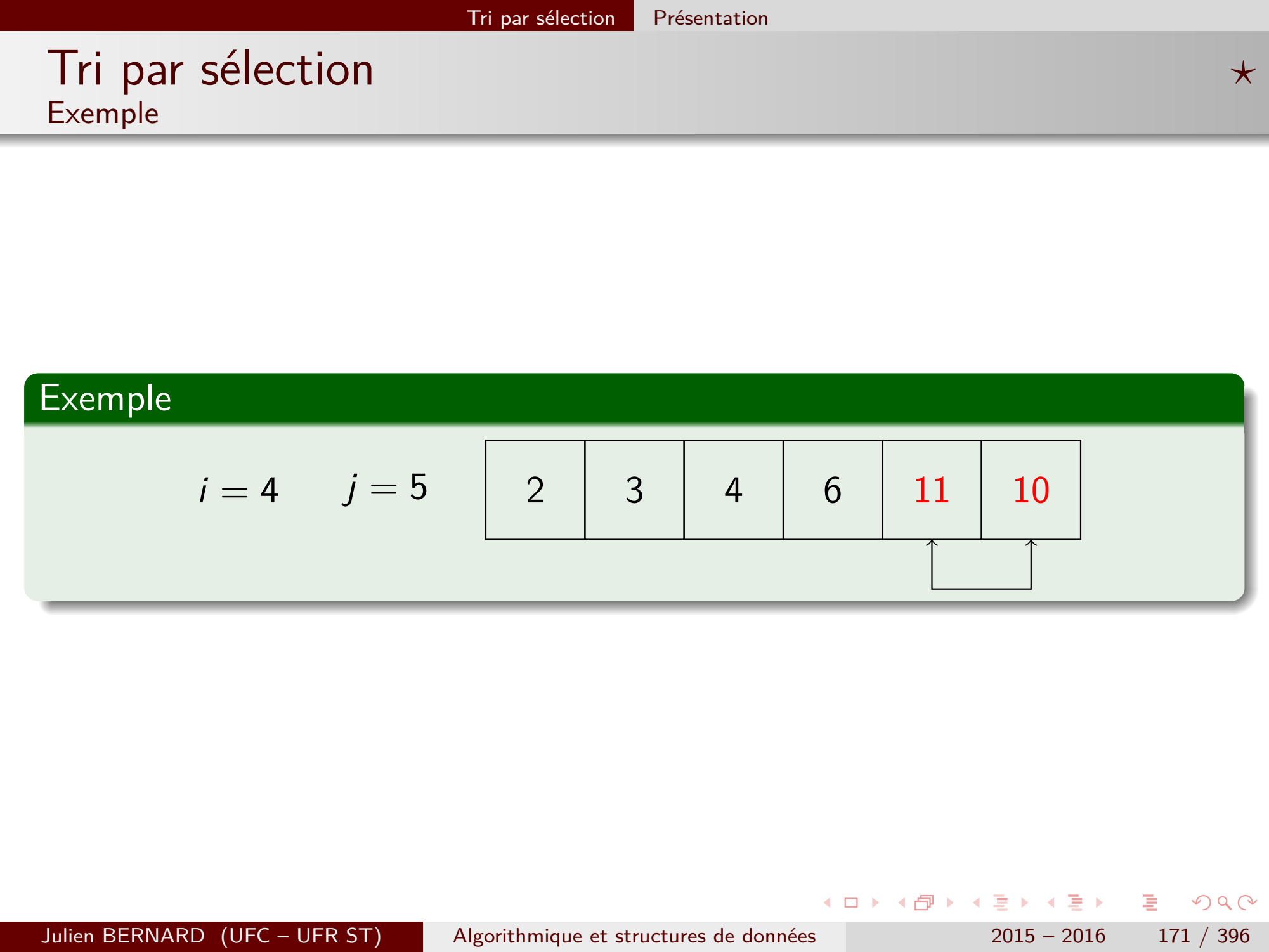
Tri par sélection
Exemple

Exemple

$i = 4$

2	3	4	6	11	10

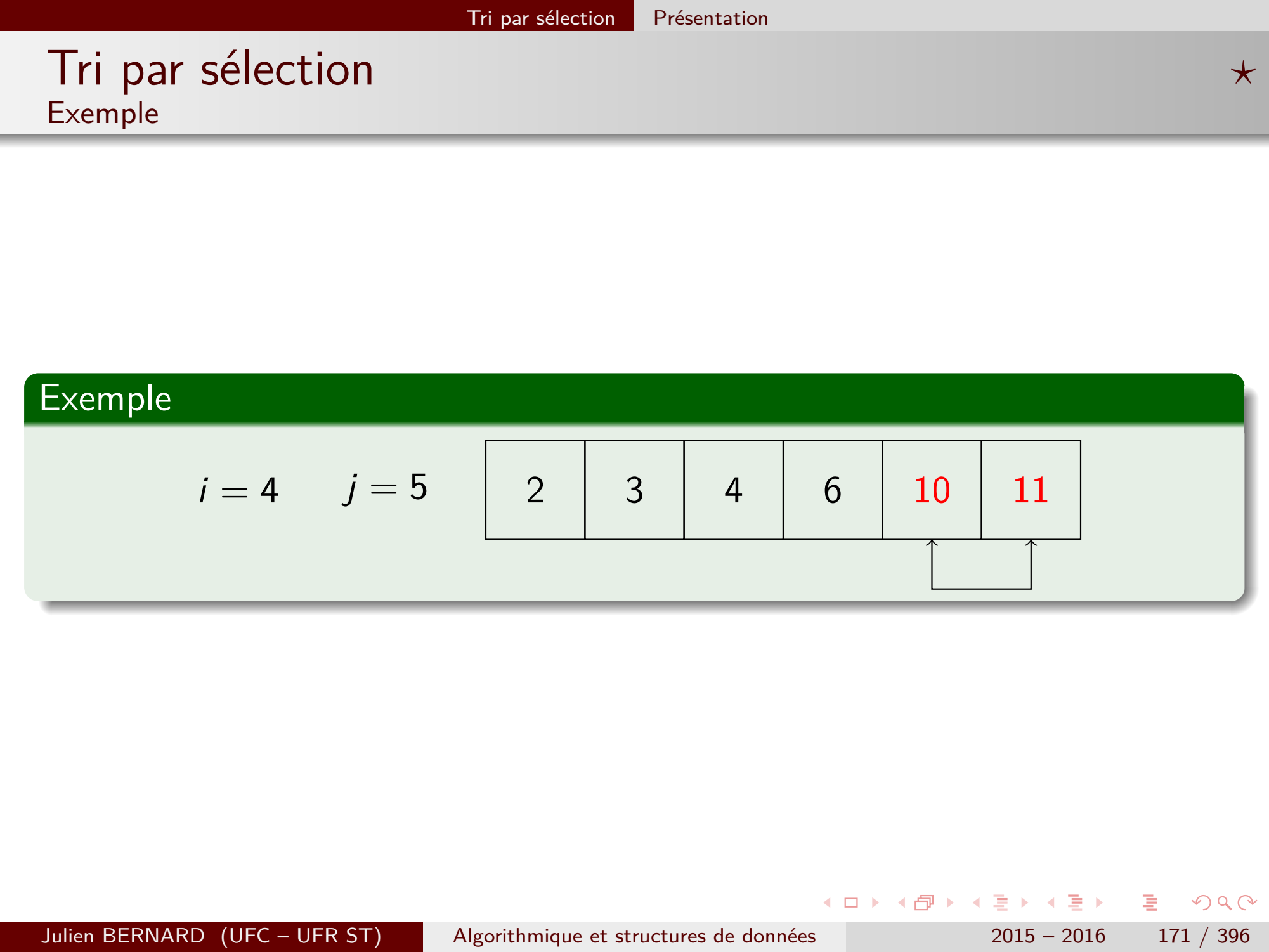
Tri par sélection
Exemple



Exemple

$i = 4 \qquad j = 5$

Tri par sélection
Exemple

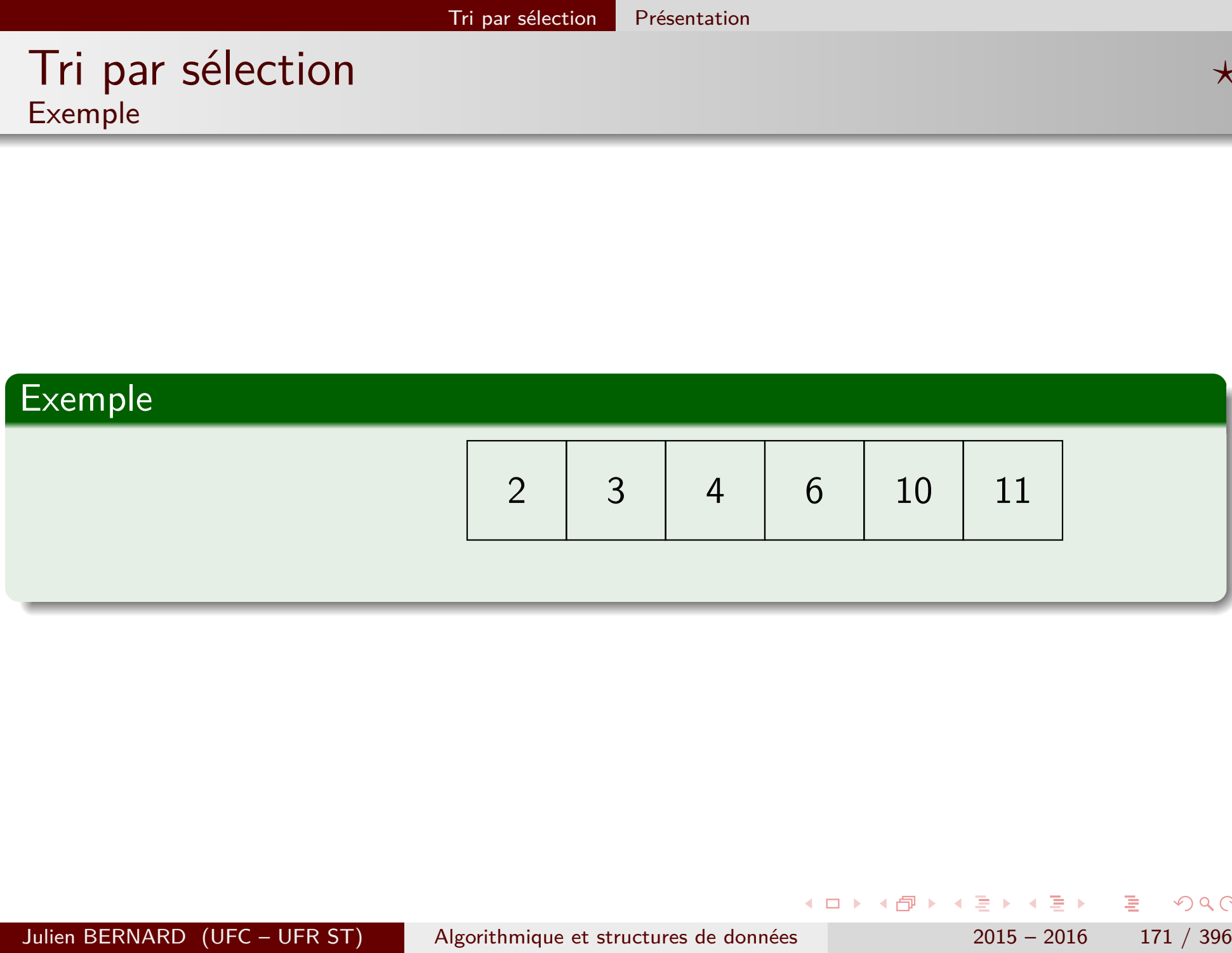


Exemple

$i = 4 \qquad j = 5$

Tri par sélection

Exemple



Exemple

2	3	4	6	10	11

Plan

Tri par sélection

Complexité en nombre de comparaisons

★★

Remarque préliminaire

Pour un tableau de taille n, cet algorithme effectue le même nombre de comparaisons, quelle que soit la valeur des éléments du tableau. Donc la complexité en moyenne et en pire cas sont les mêmes.

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

Tri par sélection

Complexité en nombre de comparaisons

★★

Complexité en nombre de comparaisons

L'algorithme effectue une comparaison pour chaque valeur de k dans la boucle interne. Cela représente $(n-1)-(i+1)+1 = n-(i+1)$ comparaisons dans la boucle interne. Il y a un appel à la boucle interne pour chaque valeur de i :

$$\mathcal{C}(n) = \sum_{i=0}^{n-2} n-(i+1)$$

Avec un changement de variable $j = n-(i+1)$, on obtient :

$$\mathcal{C}(n) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \Theta(n^2)$$

Tri par sélection

Complexité en nombre d'échanges

★★

Complexité en nombre d'échanges

Pour un tableau de taille n, cet algorithme effectue le même nombre d'échanges, quelle que soit la valeur des éléments du tableau. Donc la complexité en moyenne et en pire cas sont les mêmes.

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

L'algorithme effectue un échange par valeur de i :

$$\mathcal{C}(n) = n - 1 = \Theta(n)$$

Plan

Tri à bulles
Principe

★★★

Principe

Le *tri à bulles* consiste à faire remonter les éléments les plus légers (c'est-à-dire les plus petits) au début du tableau.
On parcourt le tableau à l'envers et on échange deux éléments consécutifs chaque fois qu'ils ne sont pas dans le bon ordre. À la fin du premier passage, le plus petit élément se retrouve en première position. On réitère en s'arrêtant à la deuxième case. Et ainsi de suite.

Tri à bulles
Algorithme

★★

Algorithme

```
void array_bubble_sort(int *data, size_t n) {
  for (size_t i = 0; i < n - 1; ++i) {
    for (size_t j = n - 1; j > i; --j) {
      if (data[j] < data[j - 1]) {
        array_swap(data, j, j - 1);
      }
    }
  }
}
```

Tri à bulles
Exemple

Exemple

$i = 0$

10	11	3	6	4	2

Tri à bulles
Exemple

Exemple

$i = 0$

10	11	3	6	4	2

Tri à bulles
Exemple

Exemple

$i = 0$

10	11	3	6	2	4

Tri à bulles
Exemple

Exemple

$i = 0$

10	11	3	2	6	4

Tri à bulles
Exemple

Exemple

$i = 0$

10	11	2	3	6	4

Tri à bulles
Exemple

Exemple

$i = 0$

10	2	11	3	6	4

Tri à bulles
Exemple

Exemple

$i = 0$

2	10	11	3	6	4

Tri à bulles
Exemple

Exemple

2	10	11	3	6	4

Tri à bulles
Exemple

Exemple

$i = 1$

2	10	11	3	6	4

Tri à bulles
Exemple

Exemple

$i = 1$

2	10	11	3	4	6

Tri à bulles
Exemple

Exemple

$i = 1$

2	10	11	3	4	6

Tri à bulles
Exemple

Exemple

$i = 1$

2	10	3	11	4	6

Tri à bulles
Exemple

Exemple

$i = 1$

2	3	10	11	4	6

Tri à bulles
Exemple

Exemple

2	3	10	11	4	6

Tri à bulles
Exemple

Exemple

$i = 2$

2	3	10	11	4	6

Tri à bulles
Exemple

Exemple

$i = 2$

2	3	10	11	4	6

Tri à bulles
Exemple

Exemple

$i = 2$

2	3	10	4	11	6

Tri à bulles
Exemple

Exemple

$i = 2$

2	3	4	10	11	6

Tri à bulles
Exemple

Exemple

2	3	4	10	11	6

Tri à bulles
Exemple

Exemple

$i = 3$

2	3	4	10	11	6

Tri à bulles
Exemple

Exemple

$i = 3$

2	3	4	10	6	11

Tri à bulles
Exemple

Exemple

$i = 3$

2	3	4	6	10	11

Tri à bulles
Exemple

Exemple

2	3	4	6	10	11

Tri à bulles
Exemple

Exemple

$i = 4$

2	3	4	6	10	11

Tri à bulles
Exemple

Exemple

$i = 4$

2	3	4	6	10	11

Tri à bulles
Exemple

Exemple

2	3	4	6	10	11

Plan

Tri à bulles

Complexité en nombre de comparaisons

★★

Remarque préliminaire

Pour un tableau de taille n, cet algorithme effectue le même nombre de comparaisons, quelle que soit la valeur des éléments du tableau. Donc la complexité en moyenne et en pire cas sont les mêmes.

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

Tri à bulles

Complexité en nombre de comparaisons

★★

Complexité en nombre de comparaisons

L'algorithme effectue une comparaison pour chaque valeur de j dans la boucle interne. Cela représente $(n-1)-(i+1)+1 = n-(i+1)$ comparaisons dans la boucle interne. Il y a un appel à la boucle interne pour chaque valeur de i :

$$\mathcal{C}(n) = \sum_{i=0}^{n-2} n-(i+1)$$

Avec un changement de variable $k = n-(i+1)$, on obtient :

$$\mathcal{C}(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2)$$

Tri à bulles

Complexité en nombre d'échanges

★★

Remarque préliminaire

Contrairement au cas précédent, le nombre d'échanges lors de l'exécution de l'algorithme du tri à bulles dépend des éléments contenus dans le tableau. On analyse donc la complexité en pire cas et la complexité en moyenne séparément.

Complexité en pire cas

Dans le pire cas, il y a un échange à chaque tour de boucle, c'est-à-dire que la condition est toujours vraie. Ce cas est obtenu quand le tableau est trié dans l'ordre décroissant. On a alors :

$$\mathcal{C}_{\text{worst}}(n) = \Theta(n^2)$$

Tri à bulles

Complexité en nombre d'échanges

★★

Complexité en moyenne

La démonstration du résultat est plus complexe que précédemment. Dans la suite, on va considérer que le tableau possède n éléments *distincts*.

Lemme

Soit deux éléments d'indice i et j, tels que $i < j$ et $t[i] > t[j]$, alors lors de l'exécution de l'algorithme, ces deux éléments seront échangés exactement une fois.

Démonstration.

Les deux éléments $t[i]$ et $t[j]$ seront échangés : au plus une fois car deux éléments dans le bon ordre ne sont pas échangés ; et au moins une fois car sinon, le tableau ne serait pas trié. □

Tri à bulles

Complexité en nombre d'échanges

★★

Définition (Tableau miroir)

Soit t un tableau de n éléments, alors on définit le *miroir* de t, qu'on note t', de la façon suivante :

$$\forall i \in [0, n-1], t'[i] = t[n-1-i]$$

Exemple

Si $t = [10, 11, 3, 6, 4, 2]$, alors $t' = [2, 4, 6, 3, 11, 10]$

Nombre d'échanges

Si on exécute l'algorithme de tri à bulles sur t et t', chaque paire d'éléments est échangée, soit dans t, soit dans t', jamais dans les deux. Donc, on a en tout autant d'échanges qu'il y a de paires d'éléments d'indices différents soit $\frac{n(n-1)}{2}$ échanges.

Tri à bulles

Complexité en nombre d'échanges

★★

Définition

On note $\mathbb{T}$ l'ensemble des tableaux qui possèdent n éléments *distincts*. On partionne $\mathbb{T}$ en deux sous-ensembles $\mathbb{T}_c$ et $\mathbb{T}_d$, tels que :

$$\mathbb{T}_c = \{t \in \mathbb{T} \mid t[0] < t[n-1]\}$$
$$\mathbb{T}_d = \{t \in \mathbb{T} \mid t[0] > t[n-1]\}$$

On suppose que tous les éléments de $\mathbb{T}$ sont équiprobables de probabilité p.

Calcul de la complexité

La complexité en moyenne est alors :

$$\mathcal{C}_{\text{avg}}(n) = \sum_{t \in \mathbb{T}} p \times \mathcal{N}(t) = p \times \sum_{t \in \mathbb{T}} \mathcal{N}(t)$$

Tri à bulles

Complexité en nombre d'échanges

★★

Calcul de la complexité (suite)

Or, on a $t \in \mathbb{T}_c$ si et seulement si $t' \in \mathbb{T}_d$, donc :

$$\begin{aligned}
\mathcal{C}_{\text{avg}}(n) &= p \times \sum_{t \in \mathbb{T}_c} \mathcal{N}(t) + p \times \sum_{t \in \mathbb{T}_d} \mathcal{N}(t) \\
&= p \times \sum_{t \in \mathbb{T}_c} \mathcal{N}(t) + p \times \sum_{t \in \mathbb{T}_c} \mathcal{N}(t') \\
&= p \times \sum_{t \in \mathbb{T}_c} (\mathcal{N}(t) + \mathcal{N}(t')) \\
&= p \times \sum_{t \in \mathbb{T}_c} \frac{n(n-1)}{2}
\end{aligned}$$

Tri à bulles

Complexité en nombre d'échanges

★★

Calcul de la complexité (suite)

$$\mathcal{C}_{\text{avg}}(n) = p \times |\mathbb{T}_c| \times \frac{n(n-1)}{2}$$

Or, comme $t \in \mathbb{T}_c$ si et seulement si $t' \in \mathbb{T}_d$, on a $|\mathbb{T}_c| = |\mathbb{T}_d| = \frac{|\mathbb{T}|}{2}$, et $p = \frac{1}{|\mathbb{T}|}$, et donc :

$$\mathcal{C}_{\text{avg}}(n) = \frac{n(n-1)}{4} = \Theta(n^2)$$

Plan

Tri par insertion

Principe

★★★

Principe

Le *tri par insertion* consiste à insérer au fur et à mesure les éléments dans la partie du tableau qui est déjà triée. C'est le tri utilisé par les joueurs de cartes.

Tri par insertion

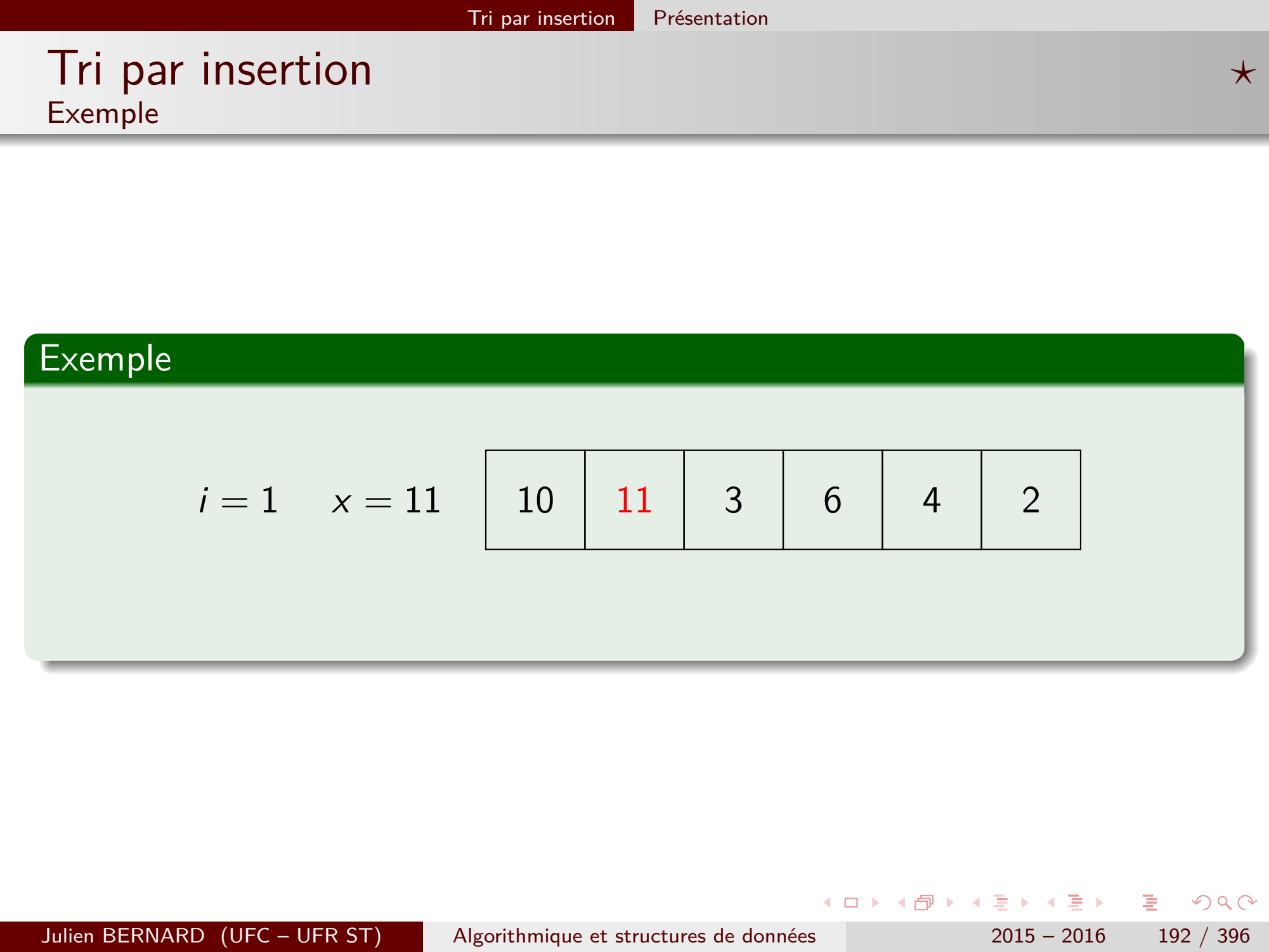
Algorithme

★★

Algorithme

```
void array_insertion_sort(int *data, size_t n) {
  for (size_t i = 1; i < n; ++i) {
    int x = data[i];
    size_t j = i;
    while (j > 0 && data[j - 1] > x) {
      data[j] = data[j - 1];
      j--;
    }
    data[j] = x;
  }
}
```

Tri par insertion
Exemple

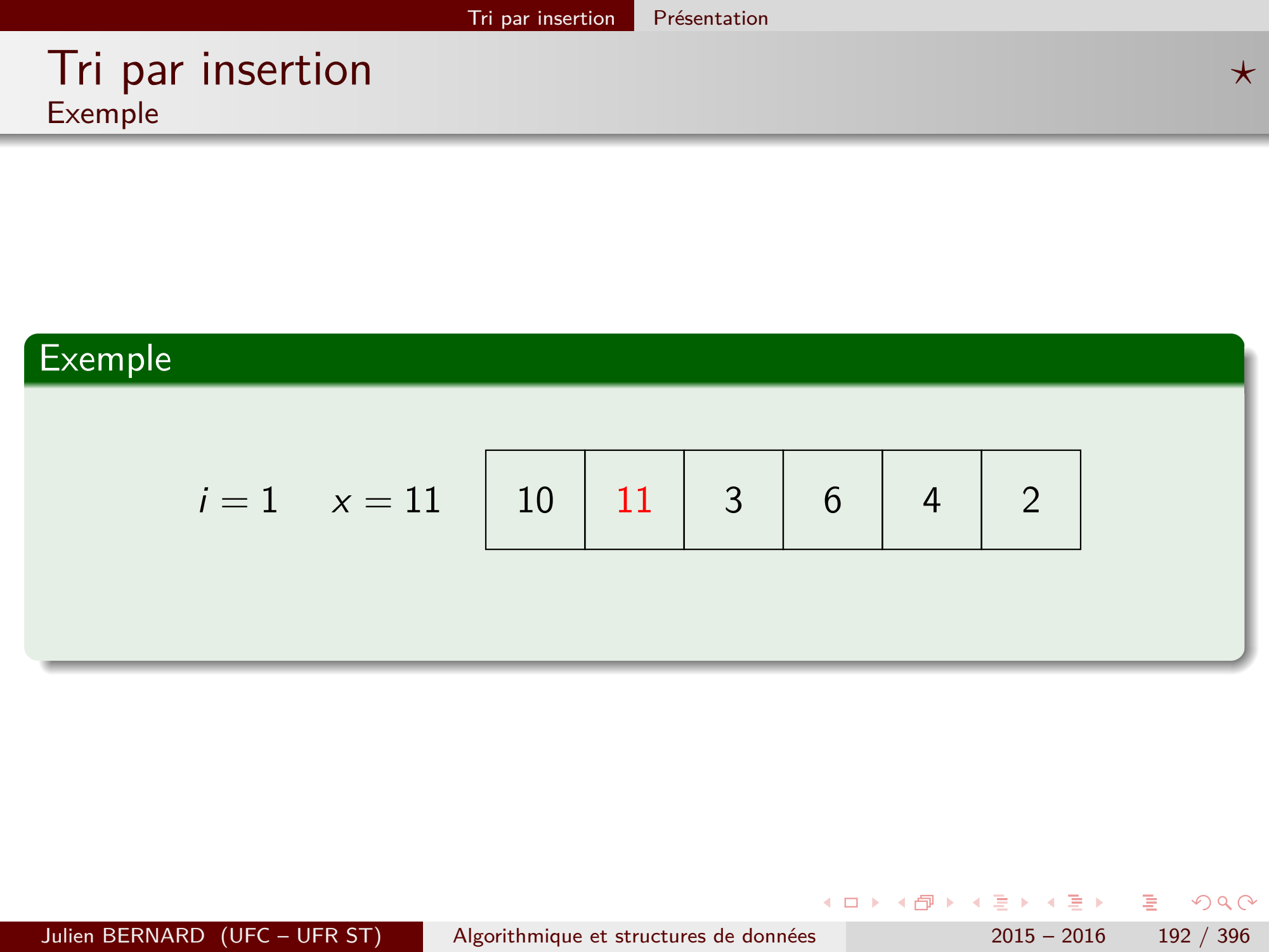


Exemple

$i = 1 \quad x = 11$

10	11	3	6	4	2

Tri par insertion
Exemple

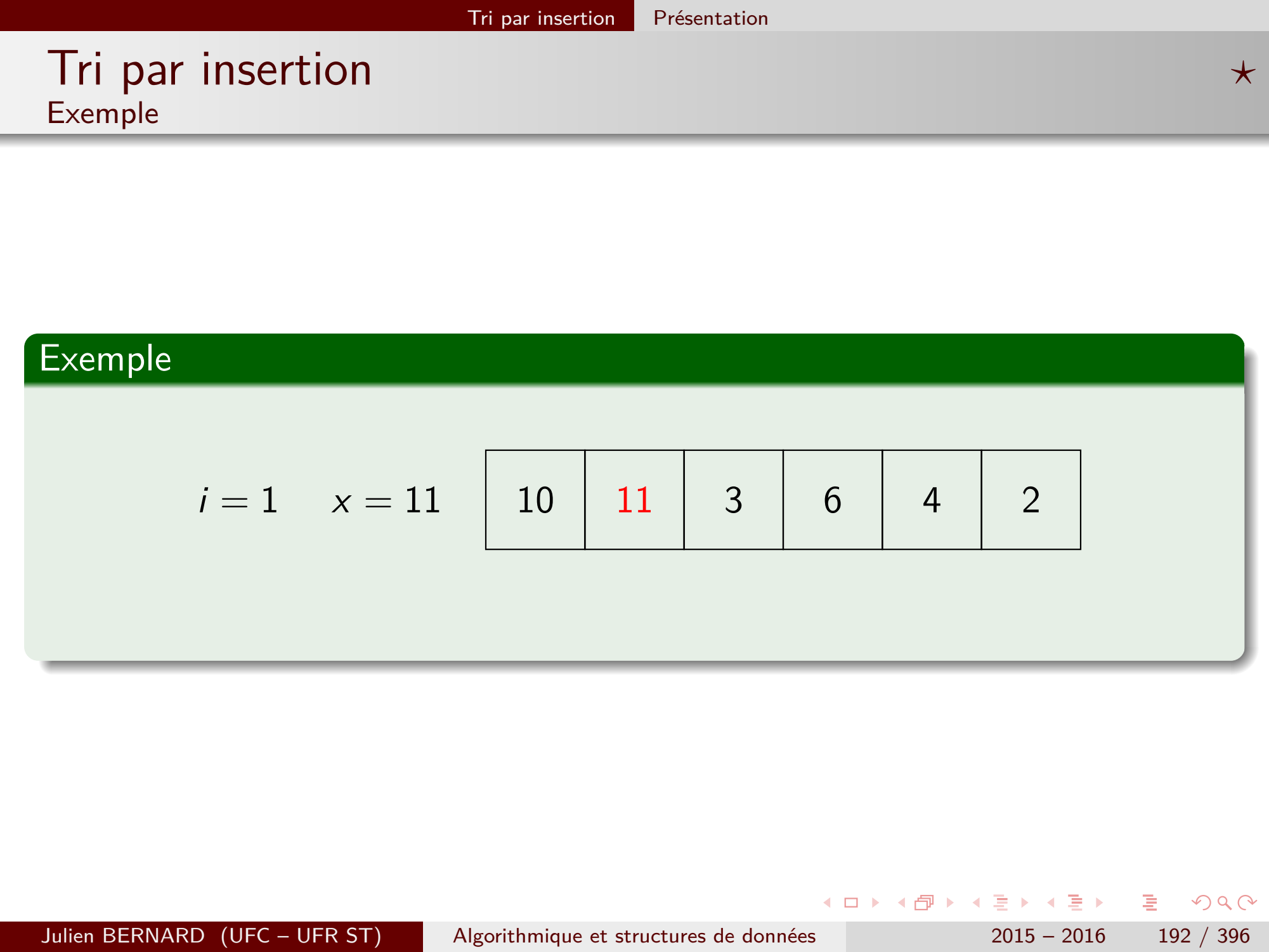


Exemple

$i = 1 \quad x = 11$

10	11	3	6	4	2

Tri par insertion
Exemple



Exemple

$i = 1 \quad x = 11$

10	11	3	6	4	2

Tri par insertion
Exemple

Exemple

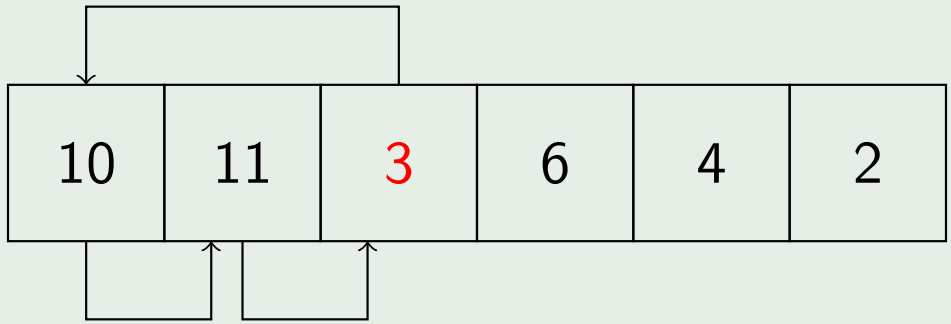
$i = 2 \qquad x = 3$

10	11	3	6	4	2

Tri par insertion
Exemple

Exemple

$i = 2 \qquad x = 3$



Tri par insertion
Exemple

Exemple

$i = 2 \qquad x = 3$

3	10	11	6	4	2

Tri par insertion
Exemple

Exemple

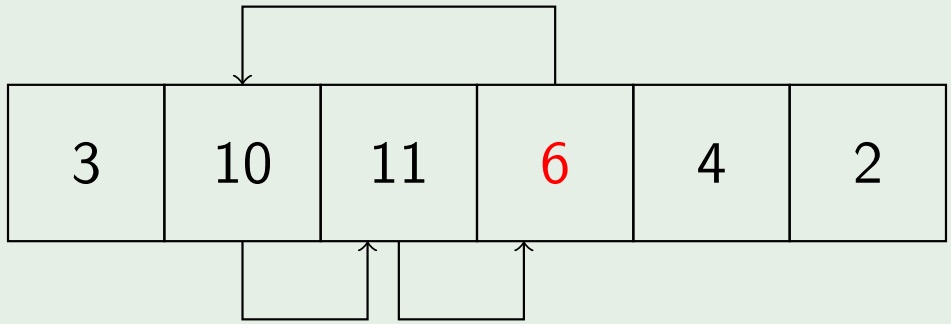
$i = 3 \qquad x = 6$

3	10	11	6	4	2

Tri par insertion
Exemple

Exemple

$i = 3 \qquad x = 6$



Tri par insertion
Exemple

Exemple

$i = 3$ $x = 6$

3	6	10	11	4	2

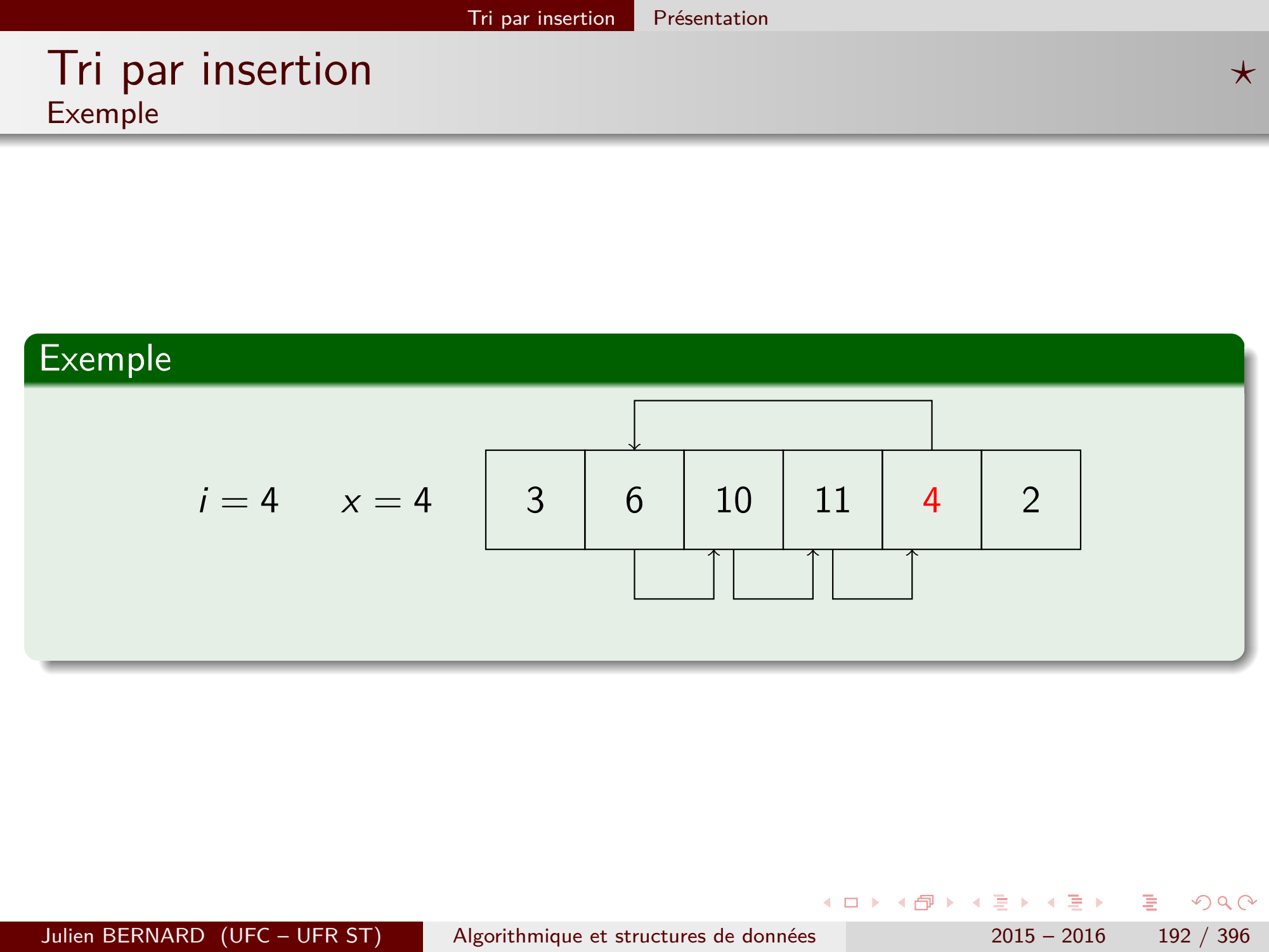
Tri par insertion
Exemple

Exemple

$i = 4 \qquad x = 4$

3	6	10	11	4	2

Tri par insertion
Exemple



Exemple

$i = 4 \qquad x = 4$

Tri par insertion
Exemple

Exemple

$i = 4 \qquad x = 4$

3	4	6	10	11	2

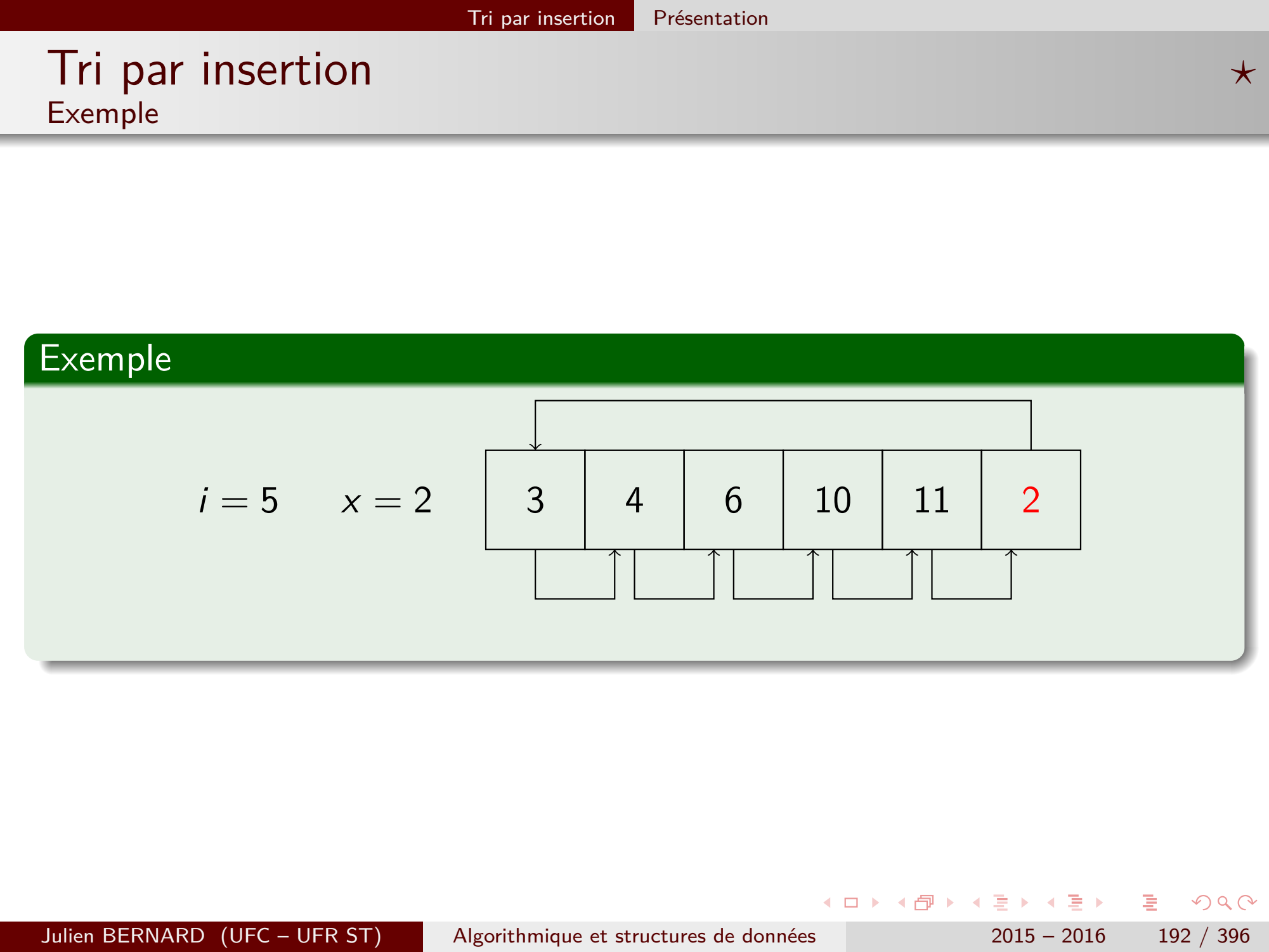
Tri par insertion
Exemple

Exemple

$i = 5 \qquad x = 2$

3	4	6	10	11	2

Tri par insertion
Exemple



Exemple

$i = 5 \qquad x = 2$

Tri par insertion
Exemple

Exemple

$i = 5 \qquad x = 2$

2	3	4	6	10	11

Tri par insertion
Exemple

Exemple

2	3	4	6	10	11

Plan

Tri par insertion

Complexité en nombre de comparaisons

Remarque préliminaire

Contrairement aux cas précédents, le nombre de comparaisons lors de l'exécution de l'algorithme du tri par insertion dépend des éléments contenus dans le tableau. On analyse donc la complexité en pire cas et la complexité en moyenne séparément.

Tri par insertion

Complexité en nombre de comparaisons

★★

Complexité en pire cas

Le pire cas est obtenu quand le tableau est trié dans l'ordre décroissant. On a alors :

$$\mathcal{C}_{\text{worst}}(n) = \Theta(n^2)$$

Complexité en moyenne

Par un raisonnement analogue au calcul de la complexité en moyenne pour le nombre d'affectations du tri à bulles, on obtient :

$$\mathcal{C}_{\text{avg}}(n) = \Theta(n^2)$$

Tri par insertion

Complexité nombre d'affectations

★★

Complexité en nombre d'affectations

Pour chaque valeur de i, le nombre d'affectations diffère au plus de 1 par rapport au nombre de comparaisons, d'où :

$$\mathcal{C}_{\text{worst}}(n) = \Theta(n^2)$$
$$\mathcal{C}_{\text{avg}}(n) = \Theta(n^2)$$

Plan

Synthèse

Synthèse

Algorithme	Comparaisons		Affectations		Stable ?	En place ?
	$\mathcal{C}_{\text{avg}}$	$\mathcal{C}_{\text{worst}}$	$\mathcal{C}_{\text{avg}}$	$\mathcal{C}_{\text{worst}}$		
Tri par sélection	$\Theta(n^2)$		$\Theta(n)$		☹	☺
Tri à bulles	$\Theta(n^2)$		$\Theta(n^2)$	$\Theta(n^2)$	☺	☺
Tri par insertion	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	☺	☺

Sixième partie

Algorithmes de tri (2/2)

Plan de ce cours

Plan

Tri rapide
Principe

★★★

Principe

Le *tri rapide* (*quicksort*) consiste à choisir un élément du tableau, appelé pivot, puis de séparer à gauche dans le tableau les éléments inférieurs au pivot et à droite dans le tableau les éléments supérieurs au pivot. Cette étape est appelée partitionnement. Puis on trie alors le sous-tableau gauche et le sous-tableau droit récursivement.

Tri rapide
Algorithme

★★

Algorithme

```
void array_quick_sort_partial(int *data,
                              ssize_t i, ssize_t j) {
  if (i < j) {
    ssize_t p = array_partition(data, i, j);
    array_quick_sort_partial(data, i, p - 1);
    array_quick_sort_partial(data, p + 1, j);
  }
}

void array_quick_sort(int *data, size_t n) {
  array_quick_sort_partial(data, 0, n - 1);
}
```

Tri rapide
Algorithme

★★

Algorithme

```
ssize_t array_partition(int *data, ssize_t i, ssize_t j) {
  ssize_t pivot_index = i;
  const int pivot = data[pivot_index];
  array_swap(data, pivot_index, j);
  ssize_t l = i;
  for (ssize_t k = i; k < j; ++k) {
    if (data[k] < pivot) {
      array_swap(data, k, l);
      l++;
    }
  }
  array_swap(data, l, j);
  return l;
}
```

Plan

Tri rapide

Complexité en nombre de comparaisons

★★★

Complexité

On note $\mathcal{C}(n)$ la complexité en nombre de comparaisons pour un tableau de taille n. On appelle k le nombre d'éléments plus petits que le pivot, alors on effectue les opérations suivantes :

- le partitionnement en $\Theta(n)$
- le tri de la partie droite en $\mathcal{C}(k)$
- le tri de la partie gauche en $\mathcal{C}(n-k-1)$

Alors, la complexité est determinée par :

$$\mathcal{C}(n) = \Theta(n) + \mathcal{C}(k) + \mathcal{C}(n-k-1)$$

Tri rapide

Complexité en nombre de comparaisons

★★★

Complexité

On a alors deux situations :

- Pire cas : $k = 0$. Alors, $\mathcal{C}(n) = \Theta(n) + \mathcal{C}(n-1)$, donc :

$$\mathcal{C}_{\text{worst}}(n) = O(n^2)$$

  Le pire cas correspond à un tableau déjà trié.

- En moyenne : $k = \frac{n}{2}$. Alors, $\mathcal{C}(n) = \Theta(n) + 2 \times \mathcal{C}(\frac{n}{2})$, donc :

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

  La complexité en moyenne est donc optimale.

Tri rapide
En pratique

Choix du pivot

Il y a plusieurs manières de choisir le pivot en pratique :

- le premier élément
- un élément au hasard
- l'élément médian entre le premier, celui du milieu et le dernier

Éléments égaux au pivot

On peut améliorer l'algorithme en prenant en compte les éléments égaux au pivot et en les plaçant à côté du pivot au milieu.

Plan

Tri fusion
Principe

★★★

Principe

Le *tri fusion* (*merge sort*) consiste à séparer le tableau en deux, puis de trier récursivement les deux moitiés, puis de fusionner les deux moitiés triées.

Remarque

Cet algorithme fonctionne mieux sur des listes que sur des tableaux. La version tableau nécessite un tableau temporaire de la taille du tableau en paramètre. Nous allons voir la version tableau.

Tri rapide
Algorithme

★★

Algorithme

```
void array_merge_sort(int *data, size_t n) {
  int *tmp = calloc(n, sizeof(int));
  array_merge_sort_partial(data, 0, n, tmp);
  free(tmp);
}
```

Tri rapide
Algorithme

★★

Algorithme

```
void array_merge_sort_partial(int *data,
                              size_t i, size_t j, int *tmp) {
  if (j - i < 2) {
    return;
  }
  size_t m = (i + j) / 2;
  array_merge_sort_partial(data, i, m, tmp);
  array_merge_sort_partial(data, m, j, tmp);
  array_merge(data, i, m, j, tmp);
  memcpy(data, tmp + i, (j - i) * sizeof(int));
}
```

Tri rapide
Algorithme

★★

Algorithme

```
void array_merge(int *data, size_t i, size_t m, size_t j,
                                                int *tmp) {
  size_t a = i;
  size_t b = m;
  for (size_t k = i; k < j; ++k) {
    if (a < m && (b == j || data[a] < data[b])) {
      tmp[k] = data[a];
      a++;
    } else {
      tmp[k] = data[b];
      b++;
    }
  }
}
```

Plan

Tri fusion

Complexité en nombre de comparaisons

★★★

Complexité

On note $\mathcal{C}(n)$ la complexité en nombre de comparaisons pour un tableau de taille n. On effectue les opérations suivantes :

- le tri de la partie droite en $\mathcal{C}\left(\frac{n}{2}\right)$
- le tri de la partie gauche en $\mathcal{C}\left(\frac{n}{2}\right)$
- la fusion en $\Theta(n)$

Alors, la complexité est determinée par :

$$\mathcal{C}(n) = 2 \times \mathcal{C}\left(\frac{n}{2}\right) + \Theta(n)$$

Et donc, en pire cas et en moyenne, la complexité est :

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

Tri fusion

En pratique

Tri en place

Il est possible d'avoir un tri fusion en place en remplaçant la fusion par une fusion en place. Toutefois, la complexité est alors en $O(n \log^2 n)$.

Listes

La version liste est toujours en $O(n \log n)$ puisqu'il n'y a pas de copie.
Mais il faut prendre garde à l'implémentation pour conserver la stabilité.

Plan

Synthèse

Synthèse

Algorithme	Comparaisons		Stable ?	En place ?
	$\mathcal{C}_{\text{avg}}$	$\mathcal{C}_{\text{worst}}$		
Tri par sélection	$\Theta(n^2)$		☹	☺
Tri à bulles	$\Theta(n^2)$		☺	☺
Tri par insertion	$\Theta(n^2)$	$\Theta(n^2)$	☺	☺
Tri rapide	$O(n \log n)$	$O(n^2)$	☹	☺
Tri fusion	$O(n \log n)$		☺	☹

Plan

En pratique

En pratique

En pratique, on utilise des combinaisons de tris. En effet, le tri par insertion est plus rapide que le tri rapide ou le tri fusion sur des tableaux jusqu'à 15–20 éléments. Donc :

- On applique un tri optimal au début (tri rapide ou tri fusion)
- Dès qu'on arrive sur des tableaux de taille inférieure à un seuil, on applique un tri par insertion

Tri rapide ou tri fusion ?

- Dans le cas de tableau, on utilisera plutôt le tri rapide qui est plus rapide que le tri fusion. Avec un choix judicieux de pivot, le pire cas arrivent suffisamment rarement.
- Dans le cas de listes chaînées, on utilise plutôt le tri fusion qui s'adapte naturellement aux listes chaînées.

Plan

Tri par dénombrement

Principe

★★★

Principe

Le *tri par dénombrement* ou *tri comptage* (*counting sort*) s'applique :

- quand les éléments sont des entiers indiscernables
- que l'intervalle des éléments $[0, m[$ est suffisamment petit.

Il consiste à compter les représentants de chaque valeur, comme dans un histogramme.

Tri par dénombrement

Algorithme

★★

Algorithme

```
void array_counting_sort(int *data, size_t n, int m) {
  size_t *hist = calloc(m, sizeof(size_t));
  for (size_t i = 0; i < n; ++i) {
    hist[data[i]] += 1;
  }
  size_t j = 0;
  for (int val = 0; val < m; ++val) {
    for (size_t k = 0; k < hist[i]; ++k) {
      data[j] = val;
      j++;
    }
  }
  free(hist);
}
```

Plan

Tri par dénombrement

Complexité

★★

Complexité

L'opération fondamentale est l'affectation (il n'y a aucune comparaison). Le nombre d'affectation pour tableau de taille n ne dépend pas du contenu du tableau. La complexité est donc de

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

Remarque

La complexité en mémoire est de $O(m)$.

Septième partie

Arbres (1/2)

Plan de ce cours

18. Généralités
    - Définitions
    - Exemples concrets
    - Parcours d'un arbre

19. Arbres binaires
    - Définitions et propriétés
    - Parcours préfixe, infixe, postfixe

Plan

Arbre
Définitions

★★★

Définition (Arbre)

Un *arbre* est :

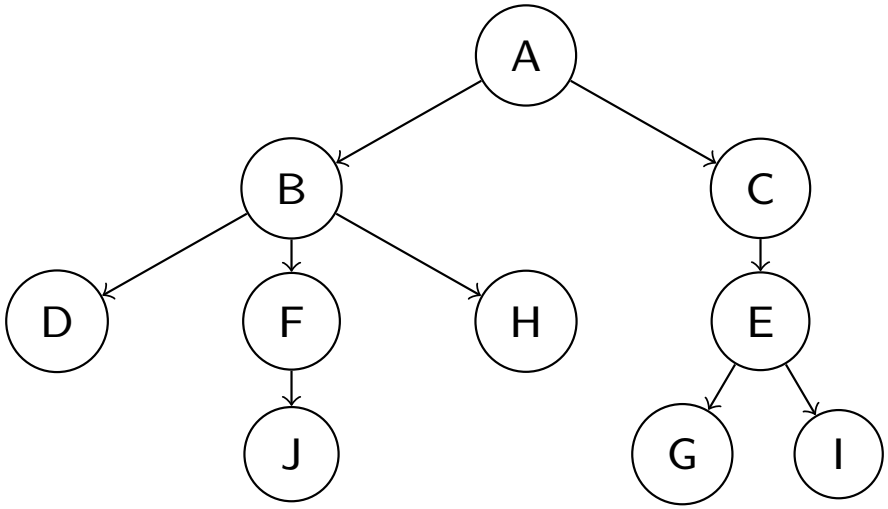
- soit l'arbre vide, noté $\varnothing$
- soit un couple (e, c), appelé *nœud*, où e est un élément, appelée parfois *étiquette*, et c est une liste d'arbres non-vides, appelés *fils*

Définition (Feuille et nœud interne)

On distingue deux types de nœud :

- une *feuille* est un nœud qui n'a pas de fils
- un *nœud interne* est un nœud qui a au moins un fils

Arbre
Définitions



Exemple (Fils, feuilles, nœuds internes)

- B et C sont les fils de A
- D, H, G et I sont des feuilles
- A, B, C, E et F sont des nœuds internes

Arbre
Définitions

★★★

Définition (Père)

Le *père* d'un nœud n est le nœud dont n est le fils.

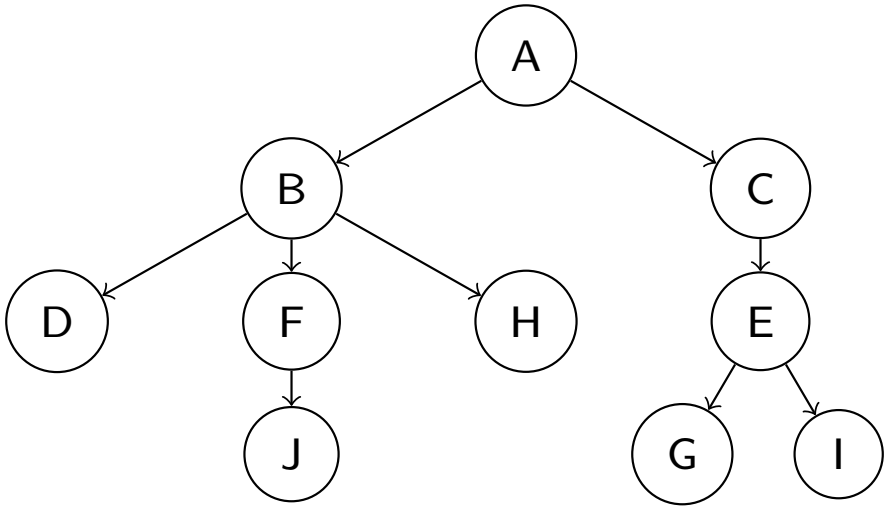
Définition (Frère)

Les *frères* d'un nœud n sont les nœuds qui ont le même père que n.

Définition (Racine)

La *racine* de l'arbre est l'unique nœud qui n'a pas de père.

Arbre
Définitions



Exemple (Père, frères, racine)

- C est le père de E
- D et H sont les frères de F
- A est la racine de l'arbre

Arbre
Définitions

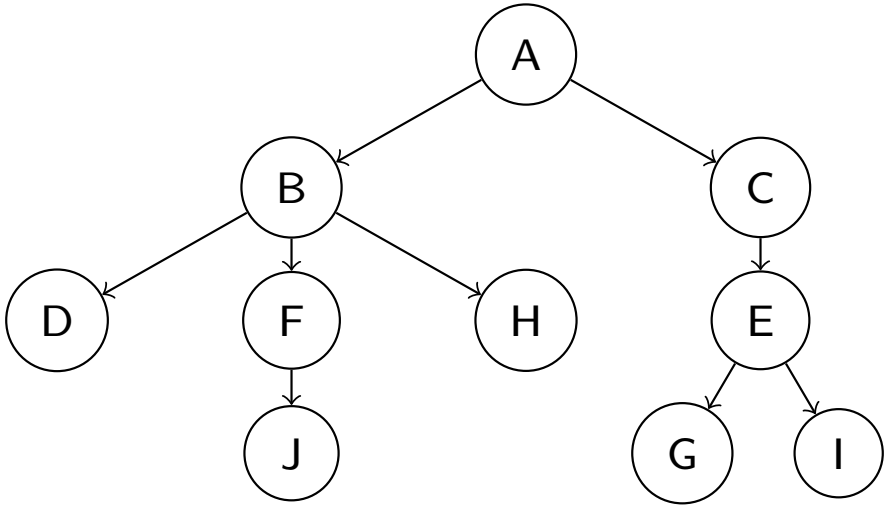
★★★

Définition (Chemin)

Le *chemin* jusqu'à un nœud n depuis la racine est la suite $[n_1, \ldots, n_p]$ où :

- n_1 est la racine
- $n_p = n$
- $\forall i \in [2, p]$, n_{i-1} est le père de n_i

Arbre
Définitions



Exemple (Chemin)

Le chemin jusqu'à G est :

$$[A, C, E, G]$$

Arbre
Définitions

★★★

Définition (Degré)

Le *degré* d'un nœud est le nombre de fils d'un nœud.

Définition (Profondeur d'un nœud)

La *profondeur* d'un nœud est le cardinal du chemin jusqu'à ce nœud.

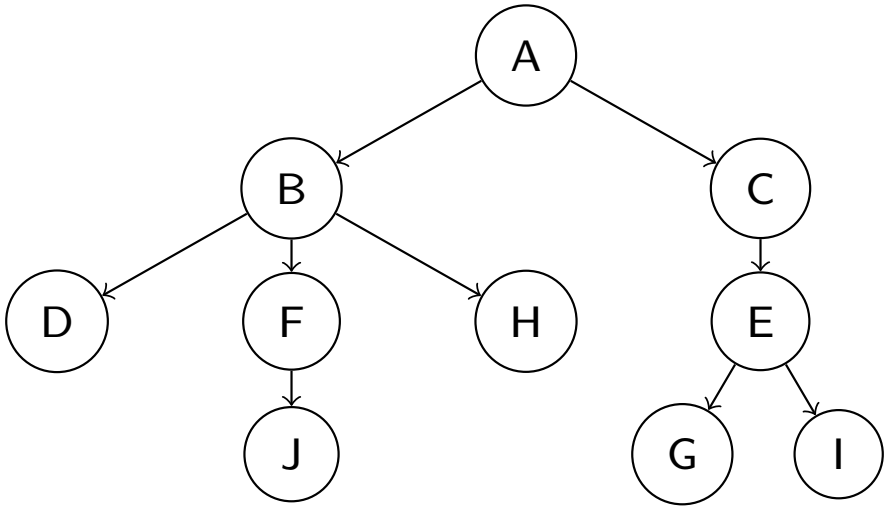
Définition (Hauteur d'un arbre)

La *hauteur* d'un arbre est la profondeur maximale des feuilles de l'arbre.

Définition (Taille d'un arbre)

La *taille* d'un arbre est le nombre de nœuds d'un arbre.

Arbre
Définitions



Exemple (Profondeur, hauteur, taille)

- B est de degré 3.
- A a une profondeur de 1
- D et E ont une profondeur de 3
- la hauteur de l'arbre est 4
- la taille de l'arbre est 10

Arbre
Définitions

★★★

Définition (Arbre *n*-aire)

Un *arbre n-aire* est un arbre dont tous les nœuds sont de degré n au plus.

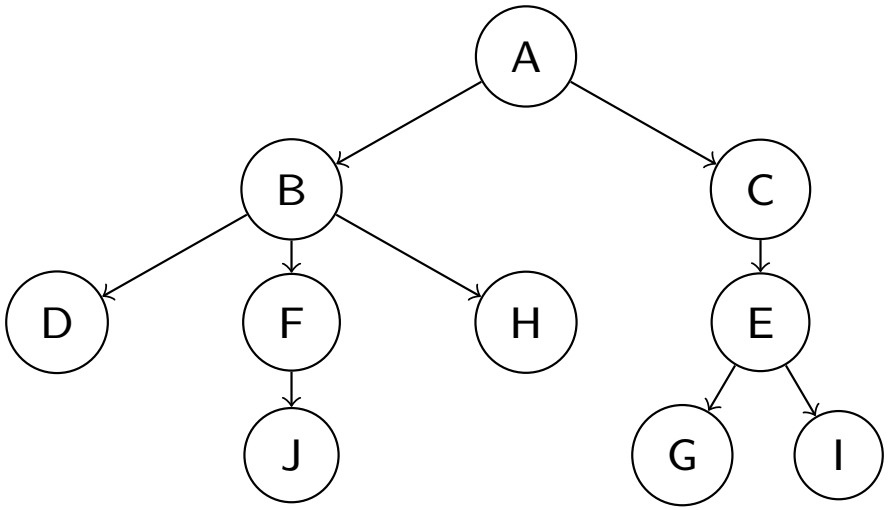
Définition (Cas particulier d'arbres *n*-aires)

- Si $n = 2$, alors on parle d'*arbre binaire*
- Si $n = 3$, alors on parle d'*arbre ternaire*

Remarque

Une liste est un arbre unaire ($n = 1$).

Arbre
Définitions



Exemple (Arbre ternaire)

- l'arbre est ternaire

Arbre
Opérations élémentaires

★★

Opérations élémentaires

- *constructeur*, noté `cons`, définie par :

$$\texttt{cons}(e, c) = (e, c)$$

- *test du vide*, noté `empty`, qui est définie par :

$$\texttt{empty}(t) = (t \stackrel{?}{=} \varnothing)$$

- *accès à l'étiquette*, noté `value`, qui est définie par :

$$\texttt{value}((e, c)) = e$$

- *accès aux fils*, noté `children`, qui est définie par :

$$\texttt{children}((e, c)) = c$$

Plan

Exemple d'arbre

Système de fichier

Exemple (Système de fichier)

- Un système de fichier est un arbre :
  - les nœuds internes sont les répertoires
  - les feuilles sont les autres fichiers
- Le nombre maximum de fils n'est pas limité, même s'il existe une limite physique.
- Dans un système Unix, la racine s'appelle /. La profondeur d'un fichier est le nombre de répertoire traversé depuis la racine plus deux (la racine et le fichier).

Exemple d'arbre
Document HTML

Exemple (Document HTML)

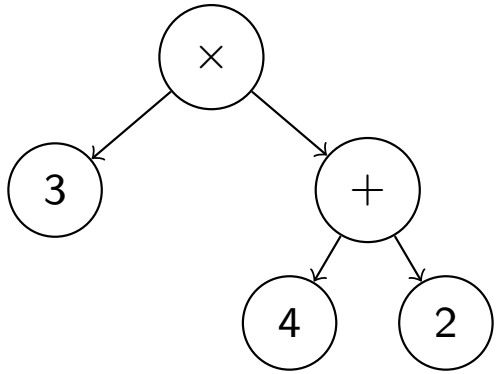
- Un document HTML est un arbre :
  - les nœuds internes sont les balises et/ou du texte
  - les feuilles sont les balises et/ou du texte
- Le nombre de fils dépend des balises.
- La racine est `<html>`

Exemple d'arbre

Expression arithmétique

Exemple (Expression arithmétique)

- Un expression arithmétique (utilisant $+$, $-$, $*$, $/$) est un arbre :
  - les nœuds internes sont les opérateurs
  - les feuilles sont les nombres
- Une expression est un arbre binaire.
- $3 \times (4 + 2)$ peut être représenté par :



Plan

Parcours d'un arbre
Définition

Définition (Parcours)

Un *parcours* d'un arbre est la visite successive de tous les nœuds (feuilles et nœuds interne) dans un certain ordre et au cours duquel on effectue une opération. On distingue deux types de parcours :

- parcours en profondeur
- parcours en largeur

Parcours d'un arbre

Parcours en profondeur

★★★

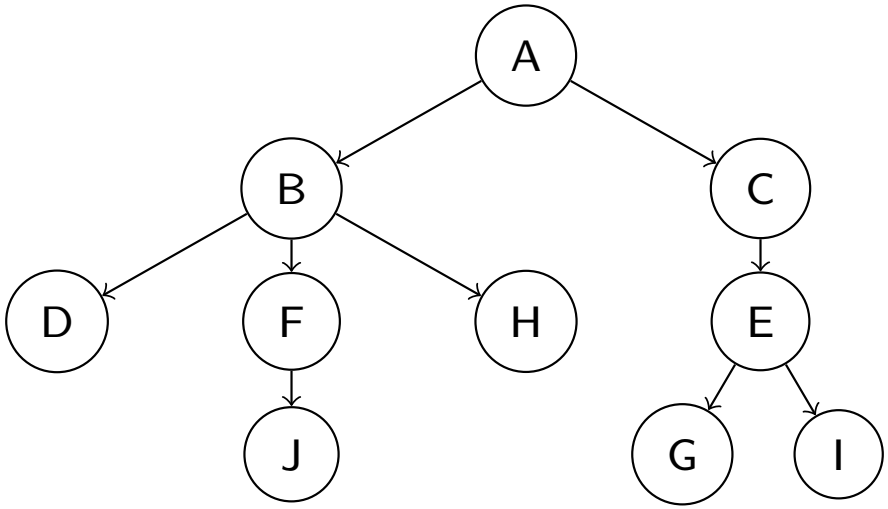
Parcours en profondeur

Le *parcours en profondeur* (*depth-first traversal*) d'un arbre permet de parcourir l'arbre de manière récursive en privilégiant les nœuds les plus profonds d'abord, c'est-à-dire qu'on va descendre dans l'arbre aussi profond que possible puis revenir et recommencer avec un autre fils, et ainsi de suite. On distingue :

- le *parcours en profondeur préfixe* où on traite l'étiquette du nœud avant de traiter les fils
- le *parcours en profondeur postfixe* où on traite l'étiquette du nœud après avoir traité les fils

Parcours d'un arbre

Parcours en profondeur



Exemple

Le parcours en profondeur préfixe de l'arbre est :

$$A \to B \to D \to F \to J \to H \to C \to E \to G \to I$$

Le parcours en profondeur postfixe de l'arbre est :

$$D \to J \to F \to H \to B \to G \to I \to E \to C \to A$$

Parcours d'un arbre

Parcours en profondeur

★★

```
function DepthFirstTraversal(t)
    if empty(t) then
        return
    end if
    ComputePreOrder(value(t))
    for c in children(t) do
        DepthFirstTraversal(c)
    end for
    ComputePostOrder(value(t))
end function
```

Parcours d'un arbre

Parcours en largeur

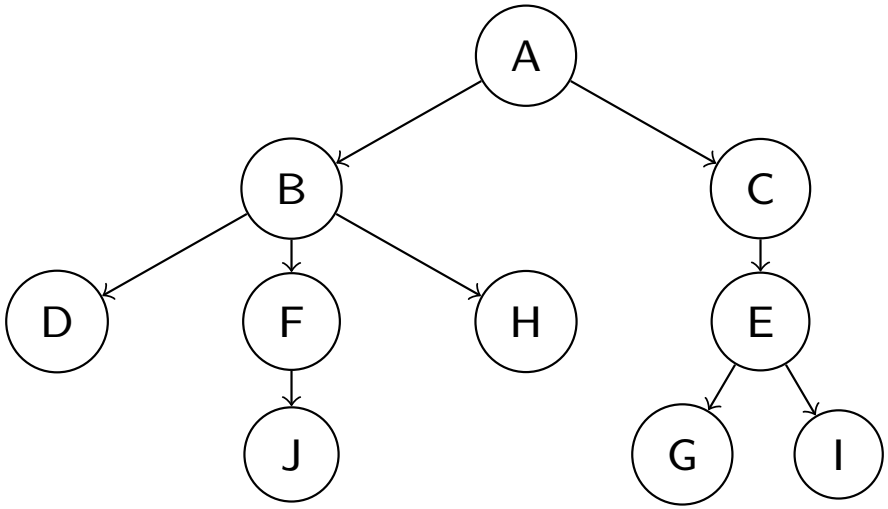
★★★

Parcours en largeur

Le *parcours en largeur* (*breadth-first traversal*) d'un arbre permet de parcourir l'arbre par ordre de profondeur croissant des nœuds, c'est-à-dire qu'on traite d'abord tous les nœuds de profondeur 1, puis tous les nœuds de profondeur 2, et ainsi de suite.

Parcours d'un arbre

Parcours en largeur



Exemple

Le parcours en largeur de l'arbre est :

$$A \to B \to C \to D \to F \to H \to E \to J \to G \to I$$

Parcours d'un arbre

Parcours en largeur

★★

```
function BreadthFirstTraversal(t)
    if empty(t) then
        return
    end if
    q ← queue()
    enqueue(q, t)
    while not empty(q) do
        u ← peek(q)
        Compute(value(u))
        for c in children(u) do
            enqueue(q, c)
        end for
        dequeue(q)
    end while
end function
```

Plan

Arbre binaire
Définition

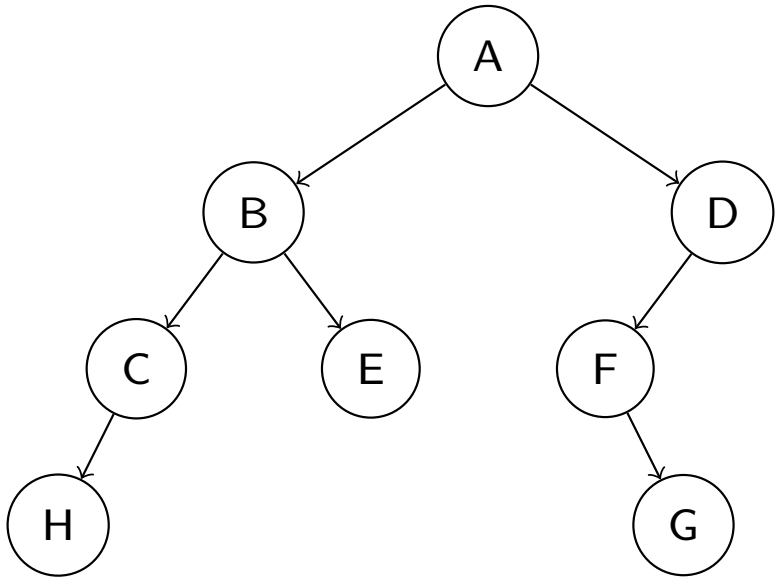
★★★

Définition (Arbre binaire)

Un *arbre binaire* est :

- soit l'arbre vide, noté $\varnothing$
- soit un triplet (e, l, r), appelé *nœud*, où e est un élément, appelée parfois *étiquette*, et l est un arbre binaire appelé *fils gauche* et r est un arbre binaire appelé *fils droit*.

Arbre binaire
Exemple



Exemple (Arbre binaire)

L'arbre ci-dessus peut être noté :

$$(A,(B,(C,(H,\varnothing,\varnothing),\varnothing),(E,\varnothing,\varnothing)),(D,(F,\varnothing,(G,\varnothing,\varnothing)),\varnothing))$$

Arbre binaire
Définitions

★★

Définition (Arbre binaire entier)

Un *arbre binaire entier* est un arbre binaire où chaque nœud a zéro ou deux fils.

Définition (Arbre binaire parfait)

Un *arbre binaire parfait* est un arbre binaire entier où toutes les feuilles ont la même profondeur.

Définition (Arbre binaire complet)

Un *arbre binaire complet* est un arbre binaire dans lequel tous les niveaux sont complets, excepté le dernier où toutes les feuilles sont rangées à gauche.

Taille et hauteur d'un arbre binaire ⋆⋆

Propriété d'un arbre binaire

Un arbre binaire de hauteur h et de taille n vérifie :

$$h \leq n \leq 2^h - 1$$

Remarques

- l'égalité à gauche a lieu pour un arbre binaire où tous les nœuds n'ont qu'un seul fils (sauf la feuille), c'est-à-dire une liste
- l'égalité à droite a lieu pour un arbre binaire parfait

Plan

Parcours dans un arbre binaire ⋆⋆⋆

Parcours dans un arbre binaire

Comme pour tout arbre, on peut effectuer des parcours en largeur et en profondeur sur un arbre binaire, où on traite les fils gauche et droit dans cet ordre. Toutefois, dans le cas d'un parcours en profondeur pour un arbre binaire, on distingue trois cas :

- le *parcours préfixe* où on traite l'étiquette du nœud avant de traiter le fils gauche
- le *parcours infixe* où on traite l'étiquette du nœud après avoir traité le fils gauche et avant de traiter le fils droit
- le *parcours postfixe* où on traite l'étiquette du nœud après avoir traité le fils droit

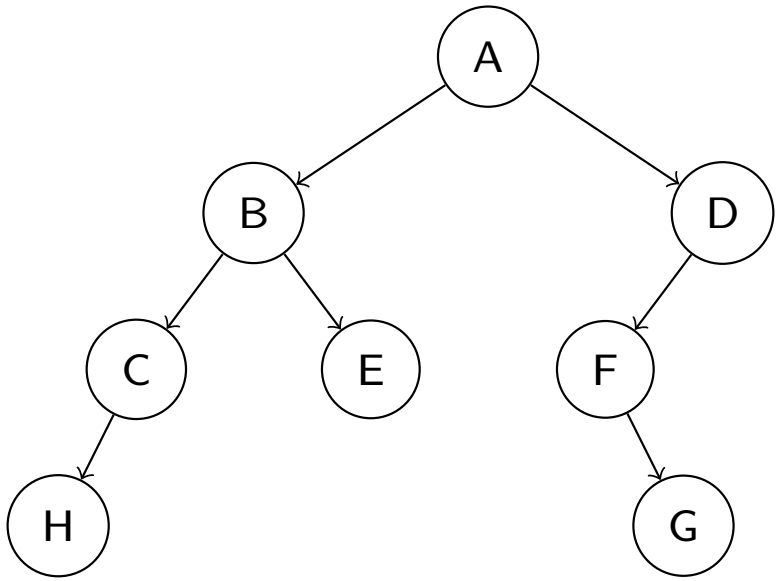
Parcours d'un arbre binaire

★★

```
function BinaryDepthFirstTraversal(t)
    if empty(t) then
        return
    end if
    ComputePreOrder(value(t))
    DepthFirstTraversal(left(t))
    ComputeInOrder(value(t))
    DepthFirstTraversal(right(t))
    ComputePostOrder(value(t))
end function
```

Parcours d'un arbre binaire

Exemple



Exemple (Parcours préfixe, infixe, postfixe)

Le parcours préfixe est : $A \to B \to C \to H \to E \to D \to F \to G$

Le parcours infixe est : $H \to C \to B \to E \to A \to F \to G \to D$

Le parcours postfixe est : $H \to C \to E \to B \to G \to F \to D \to A$

Huitième partie

Arbres (2/2)

Plan de ce cours

20 Arbres binaires de recherche
- Définitions
- Opérations sur les arbres binaires de recherche

21 Tas
- Définition
- Opérations sur les tas
- Tri par tas

Plan

Arbre binaire de recherche

Définition

★★★

Définition (Arbre binaire de recherche)

Soit E un ensemble muni d'une relation d'ordre. Un *arbre binaire de recherche* (*binary search tree*) d'éléments de E est un arbre binaire tel que :

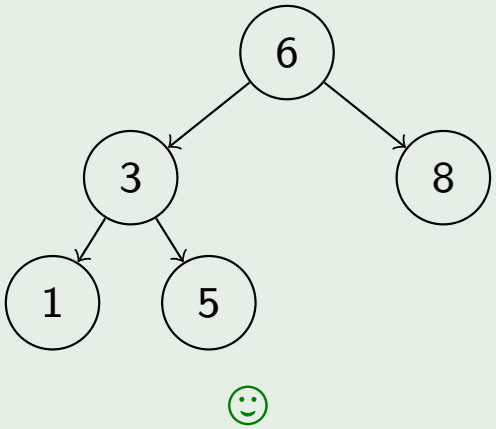
- tous les nœuds de son sous-arbre gauche sont plus petits que sa racine, suivant la relation d'ordre ;
- tous les nœuds de son sous-arbre droit sont plus grands que sa racine, suivant la relation d'ordre ;
- son sous-arbre gauche et son sous-arbre droit sont des arbres binaires de recherche.

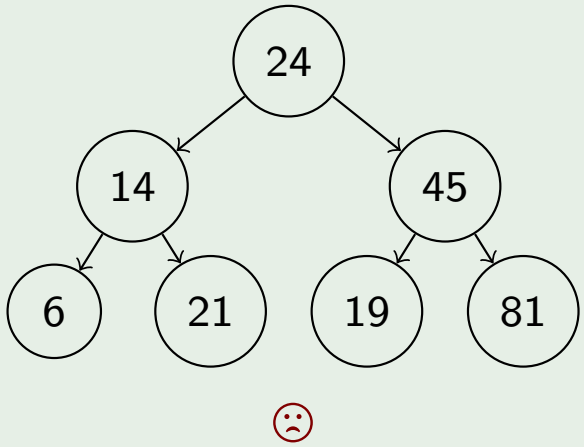
Arbre binaire de recherche
Exemples

Exemples (Ensembles E munis d'une relation d'ordre)

- l'ensemble des entiers, avec $<$ comme relation d'ordre
- l'ensemble des réels, avec $<$ comme relation d'ordre
- l'ensemble des chaînes de caractères, avec l'ordre lexicographique comme relation d'ordre

Exemples (Arbres binaires de recherche)





Arbre binaire de recherche

Remarque

★★

Remarque

Il existe deux types d'arbre binaire de recherche :

- les arbres binaires de recherche où on n'autorise pas les éléments égaux, ces arbres sont utilisés pour représenter des ensembles mathématiques (*set*) ;
- les arbres binaires de recherche où on autorise les éléments égaux, ces arbres sont utilisés pour représenter des multiensembles (*multiset*) ou sac (*bag*) ;

Parcours infixe

Le parcours infixe d'un arbre binaire de recherche rencontre l'ensemble des éléments dans l'ordre croissant.

Arbre binaire de recherche

Implémentation

★★

Implémentation

```
struct bst {
  int data;
  struct bst *left;
  struct bst *right;
};
```

Remarque

Chaque opération va devoir maintenir la structure en arbre binaire de recherche.

Plan

Opérations sur les arbres binaires de recherche ⋆⋆

Opérations sur les arbres binaires de recherche

On définit trois opérations sur les arbres binaires de recherche

- la recherche d'un élément ;
- l'insertion d'un élément ;
- la suppression d'un élément.

Pour un arbre binaire de taille n et de hauteur h, on exprimera la complexité de ces opérations en fonction de n et h, on n'ouliant pas la relation :

$$h \leq n \leq 2^h - 1$$

Opérations sur les arbres binaires de recherche

Recherche d'un élément

★★★

Principe

Pour la *recherche* d'un élément e dans un arbre binaire de recherche, on utilise la propriété des arbres binaires de recherche et on va descendre dans l'arbre en comparant e à l'étiquette e' de chaque nœud.

- si $e = e'$, alors on a trouvé l'élément et la recherche a réussi ;
- si $e < e'$, alors e, s'il est présent, est dans le sous-arbre gauche ;
- si $e > e'$, alors e, s'il est présent, est dans le sous-arbre droit ;
- si on arrive sur un arbre vide, alors la recherche a échoué.

Opérations sur les arbres binaires de recherche

Recherche d'un élément

★★

Algorithme

```
bool bst_search(const struct bst *self, int data) {
  if (self == NULL) {
    return false;
  }
  if (data < self->data) {
    return bst_search(self->left, data);
  }
  if (data > self->data) {
    return bst_search(self->right, data);
  }
  return true;
}
```

Opérations sur les arbres binaires de recherche

Recherche d'un élément

★★

Complexité

L'opération fondamentale est la comparaison ($<$ et $>$). La fonction `bst_search` parcourt au plus une branche de l'arbre, donc au plus h nœuds. Elle fait au maximum deux comparaisons par nœud visité, c'est-à-dire $O(1)$. Donc, la complexité pour un arbre de taille n est :

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

Deux cas se présentent :

- si l'arbre est équilibré, alors $\mathcal{C}(n) = O(\log n)$
- si l'arbre est déséquilibré, alors $\mathcal{C}(n) = O(n)$

Opérations sur les arbres binaires de recherche

Insertion d'un élément

★★★

Principe

Pour l'*insertion* d'un élément *e* dans un arbre binaire de recherche, on va descendre dans l'arbre pour insérer *e* dans une feuille (s'il n'est pas déjà présent dans l'abre). La descente dans l'arbre est quasi-identique à la recherche d'un élément dans un arbre binaire de recherche.

Opérations sur les arbres binaires de recherche

Insertion d'un élément

★★

Algorithme

```
struct bst *bst_insert(struct bst *self, int data) {
  if (self == NULL) {
    struct bst *tree = malloc(sizeof(struct bst));
    tree->left = tree->right = NULL; tree->data = data;
    return tree
  }
  if (data < self->data) {
    self->left = bst_insert(self->left, data);
    return self->left;
  }
  if (data > self->data) {
    self->right = bst_insert(self->right, data);
    return self->right;
  }
  return self;
}
```

Opérations sur les arbres binaires de recherche

Insertion d'un élément

★★

Complexité

L'opération fondamentale est la comparaison ($<$ et $>$). La fonction `bst_insert` parcourt une branche de l'arbre, donc au plus h nœuds. Elle fait au maximum deux comparaisons par nœud visité, c'est-à-dire $O(1)$. Donc, la complexité pour un arbre de taille n est :

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

Deux cas se présentent :

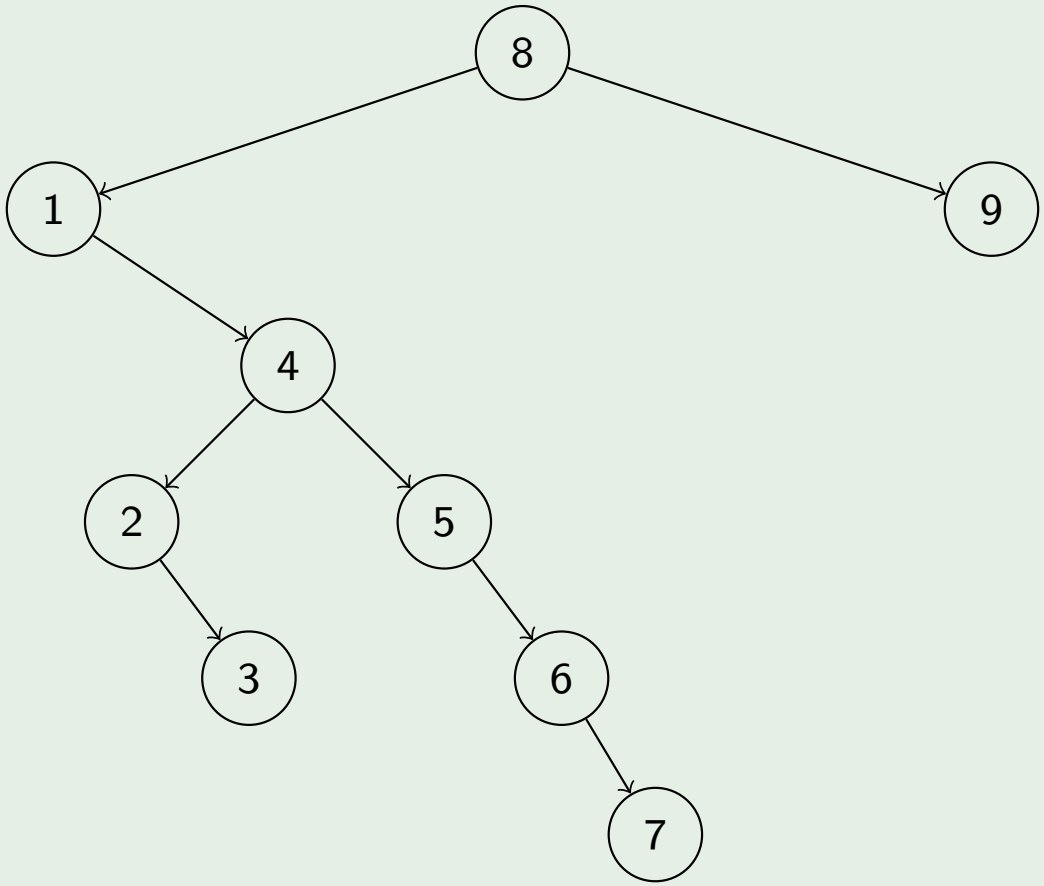
- si l'arbre est équilibré, alors $\mathcal{C}(n) = O(\log n)$
- si l'arbre est déséquilibré, alors $\mathcal{C}(n) = O(n)$

Opérations sur les arbres binaires de recherche

Insertion d'un élément

Exemple (Insertion dans un arbre binaire de recherche)

Arbre binaire de recherche après avoir inséré : $8, 1, 4, 2, 5, 9, 3, 6, 7$



Opérations sur les arbres binaires de recherche

Suppression d'un élément

★★★

Principe

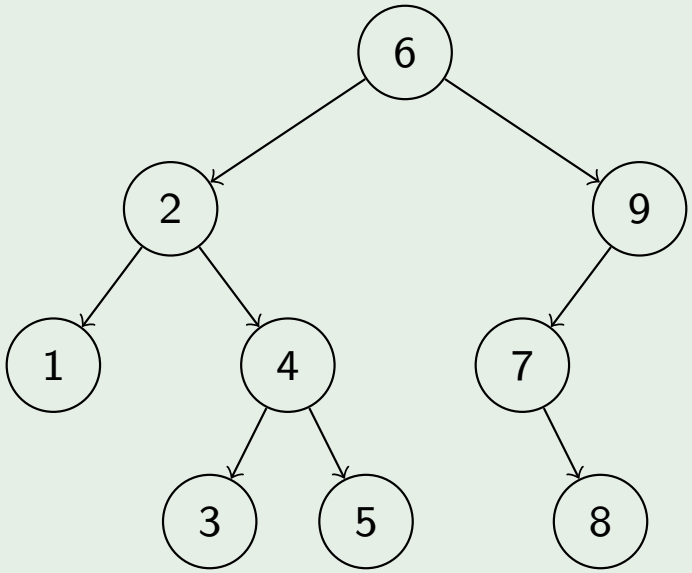
Pour la *suppression* d'un élément e dans un arbre binaire de recherche, on commence par chercher cet élément dans l'arbre, puis :

- si le nœud est une feuille, on supprime la feuille ;
- si le nœud a un fils unique, on remplace le nœud par ce fils ;
- si le nœud a deux fils, on remplace la valeur du nœud par le minimum du sous-arbre droit qu'on enlève du sous-arbre droit.

Opérations sur les arbres binaires de recherche

Suppression d'un élément

Exemple (Suppression dans un arbre binaire de recherche)



Opérations sur les arbres binaires de recherche ⋆⋆

Suppression d'un élément

Remarque

Le minimum du sous-arbre droit est toujours l'élément le plus à gauche, il n'a donc pas de fils gauche. Il se supprime donc facilement en le remplaçant par son fils droit (s'il existe).

Algorithme

```
struct bst *bst_delete_minimum(struct bst *self, struct bst **min) {
  if (self->left == NULL) {
    struct bst *right = self->right;
    self->right = NULL;
    *min = self;
    return right;
  }
  self->left = bst_delete_minimum(self->left, min);
  return self;
}
```

Opérations sur les arbres binaires de recherche ⋆⋆

Suppression d'un élément

Algorithme

```
struct bst *bst_delete(struct bst *self) {
  struct bst *left = self->left;
  struct bst *right = self->right;
  free(self); self = NULL;
  if (left == NULL && right == NULL) {
    return NULL;
  }
  if (left == NULL) {
    return right;
  }
  if (right == NULL) {
    return left;
  }
  right = bst_delete_minimum(right, &self);
  self->left = left; self->right = right;
  return self;
}
```

Opérations sur les arbres binaires de recherche

Suppression d'un élément

★★

Algorithme

```
struct bst *bst_remove(struct bst *self, int data) {
  if (self == NULL) {
    return NULL;
  }
  if (data < self->data) {
    self->left = bst_remove(self->left, data);
    return self;
  }
  if (data > self->data) {
    self->right = bst_remove(self->right, data);
    return self;
  }
  return bst_delete(self);
}
```

Opérations sur les arbres binaires de recherche

Suppression d'un élément

★★

Complexité

L'opération fondamentale est la comparaison ($<$ et $>$). Les fonctions `bst_remove`, `bst_delete`, `bst_delete_minimum` parcourent une branche de l'arbre, donc au plus h nœuds. Elles font au maximum deux comparaisons par nœud visité, c'est-à-dire $O(1)$. Donc, la complexité pour un arbre de taille n est :

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

Deux cas se présentent :

- si l'arbre est équilibré, alors $\mathcal{C}(n) = O(\log n)$
- si l'arbre est déséquilibré, alors $\mathcal{C}(n) = O(n)$

Opérations sur les arbres binaires de recherche

Synthèse

★★

Synthèse

Opération	Arbre équilibré	Arbre déséquilibré
Recherche	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Suppression	$O(\log n)$	$O(n)$

Question

Peut-on s'assurer que les arbres binaires de recherche restent équilibrés ?

Remarque

La fonction de recherche ne modifie pas l'arbre donc, elle est identique pour tous les arbres binaires de recherche.

Arbres binaires équilibrés ⋆

Arbres binaires équilibrés

Il existe des variantes des arbres binaires de recherche qui permettent de conserver l'équilibre et donc d'assurer l'optimalité des opérations :

- les arbres AVL
- les arbres bicolores ou arbres rouge-noir

Dans les deux cas, pour chaque insertion ou suppression, on réalise des *rotations* pour conserver l'équilibre général de l'arbre.

Plan

Tas
Définitions

★★★

Définition (Arbre binaire complet)

Un *arbre binaire complet* est un arbre binaire dans lequel tous les niveaux sont complets, excepté le dernier où toutes les feuilles sont rangées à gauche.

Définition (Arbre partiellement ordonné)

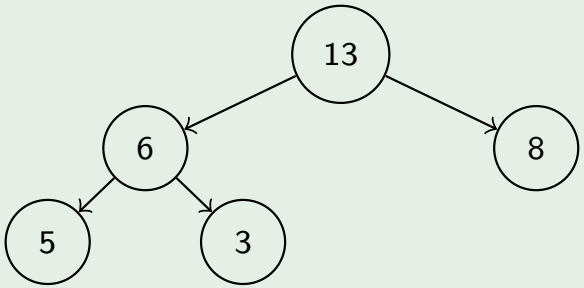
Un *arbre partiellement ordonné* est un arbre dont l'étiquette de chaque nœud est supérieure ou égale à l'étiquette de ses fils.

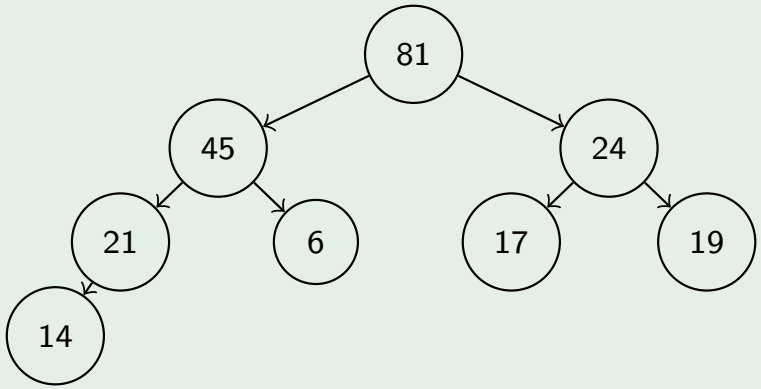
Définition (Tas)

Un *tas* (*heap*) est un arbre binaire complet partiellement ordonné.

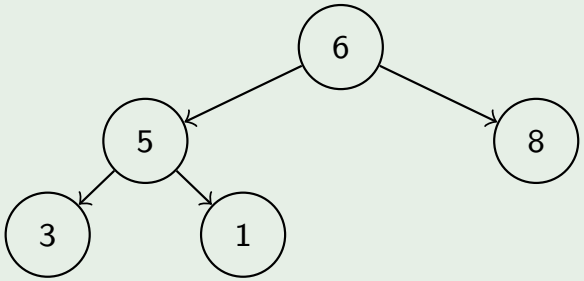
Tas
Exemples

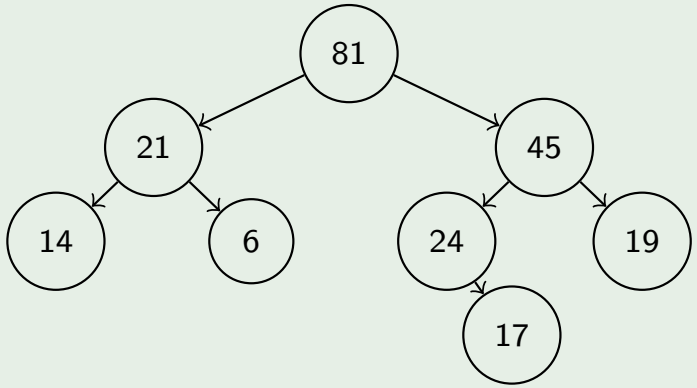
Exemples (Tas)





Exemples (Non-tas)





Tas
Représentation

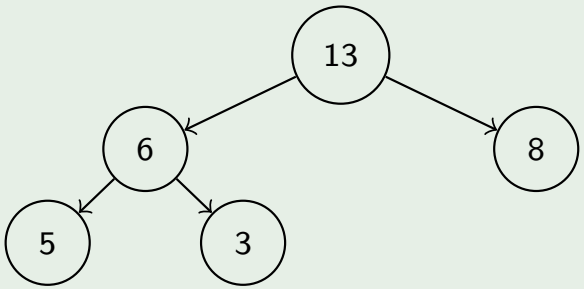
★★

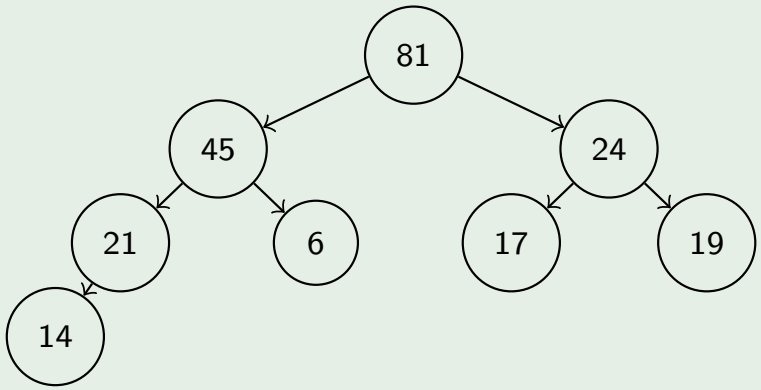
Représentation

On représente un tas à l'aide d'un tableau. Il suffit de lire les nœuds de haut en bas et de gauche à droite.

Exemples (Représentation d'un tas par un tableau)

Les tas suivants sont représentés respectivement par les tableaux $[13, 6, 8, 5, 3]$ et $[81, 45, 24, 21, 6, 17, 19, 14]$.





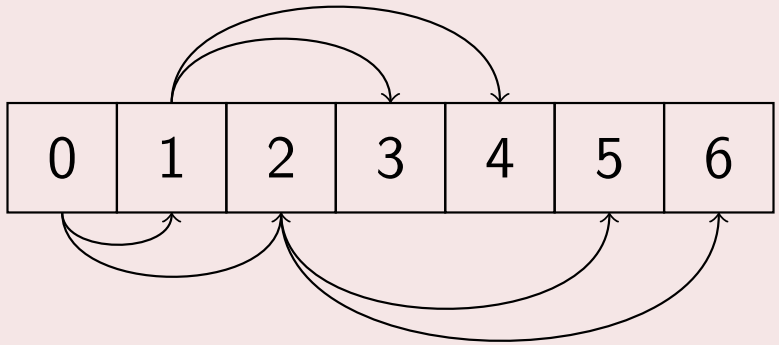
Tas
Représentation

★★

Représentation

Si h est un tableau de taille p qui représente un tas :

- $h[0]$ est le *sommet* du tas, c'est-à-dire la racine de l'arbre
- $h[i]$ a pour père $h[\frac{i-1}{2}]$
- $h[i]$ a pour fils gauche $h[2 \times i + 1]$ (s'il existe)
- $h[i]$ a pour fils droit $h[2 \times i + 2]$ (s'il existe)
- si $i \geq \frac{p}{2}$ alors $h[i]$ est une feuille



Tas
Utilisation

Utilisation

Hormis le tri par tas, les tas sont utilisés pour représenter des *files de priorité* (*priority queue*), c'est-à-dire des files dans laquelle on peut accéder à l'élement le plus prioritaire.

Plan

Opérations sur les tas ⋆⋆

Opérations sur les tas

On définit deux opérations sur les tas :

- l'insertion d'un élément ;
- la suppression du sommet du tas.

Opérations sur les tas

Insertion d'un élément

★★★

Principe

Pour l'*insertion* d'un élément e dans un tas :

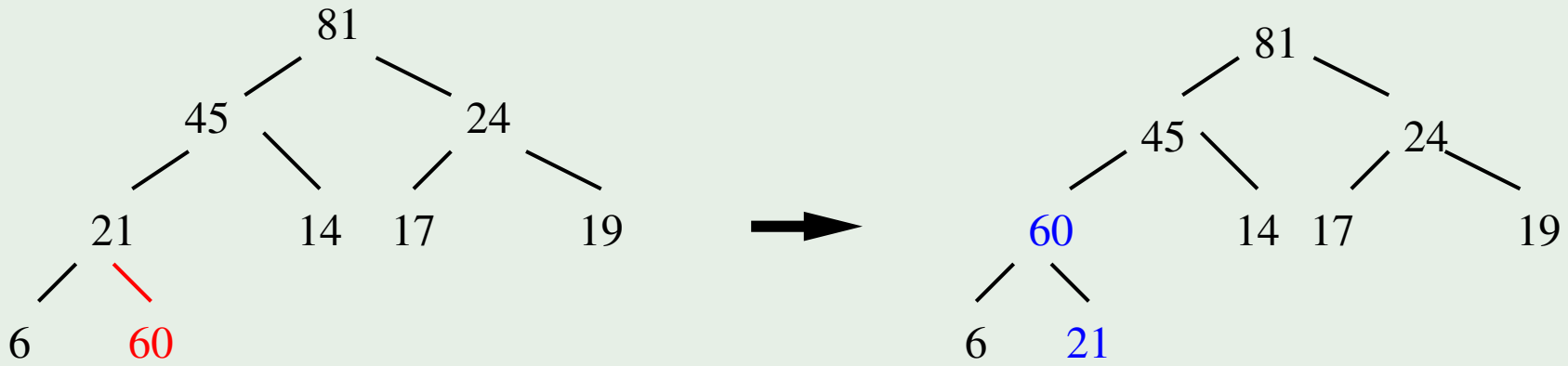
1. on ajoute l'élément au dernier niveau de l'arbre ou au niveau suivant si celui-ci est complet, ce qui maintient l'arbre complet ;
2. on fait remonter la feuille en l'échangeant avec son père s'il est plus petit, puis à nouveau avec son père s'il est plus petit, et ainsi de suite, ce qui maintient l'arbre partiellement ordonné.

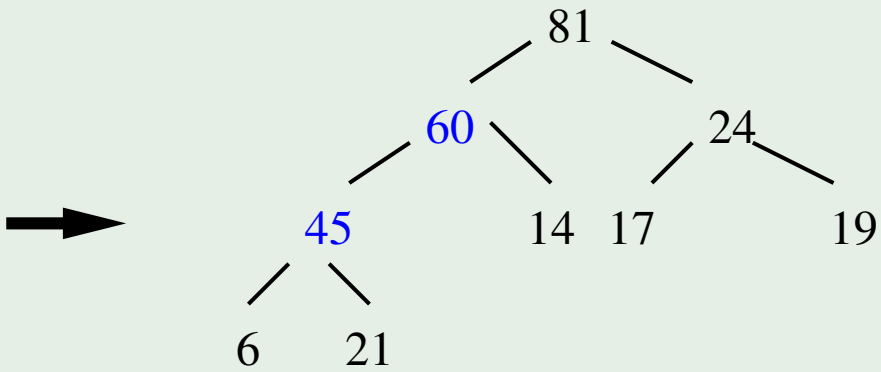
Opérations sur les tas

Insertion d'un élément

★★

Exemples (Insertion d'un élément)





Opérations sur les tas

Insertion d'un élément

Algorithme

```
void heap_insert(int *heap, size_t n, int value) {
  size_t i = n;
  heap[i] = value;
  while (i > 0) {
    ssize_t j = (i - 1) / 2;
    if (heap[i] < heap[j]) {
      break;
    }
    array_swap(heap, i, j);
    i = j;
  }
}
```

Opérations sur les tas

Insertion d'un élément

★★

Complexité

L'opération fondamentale est la comparaison ($>$). La fonction `heap_insert` parcourt une branche de l'arbre, donc au plus h nœud. Elle fait une comparaison par nœud visité. Donc, la complexité pour un tas de taille n est :

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

Opérations sur les tas
Suppression du sommet

★★★

Principe

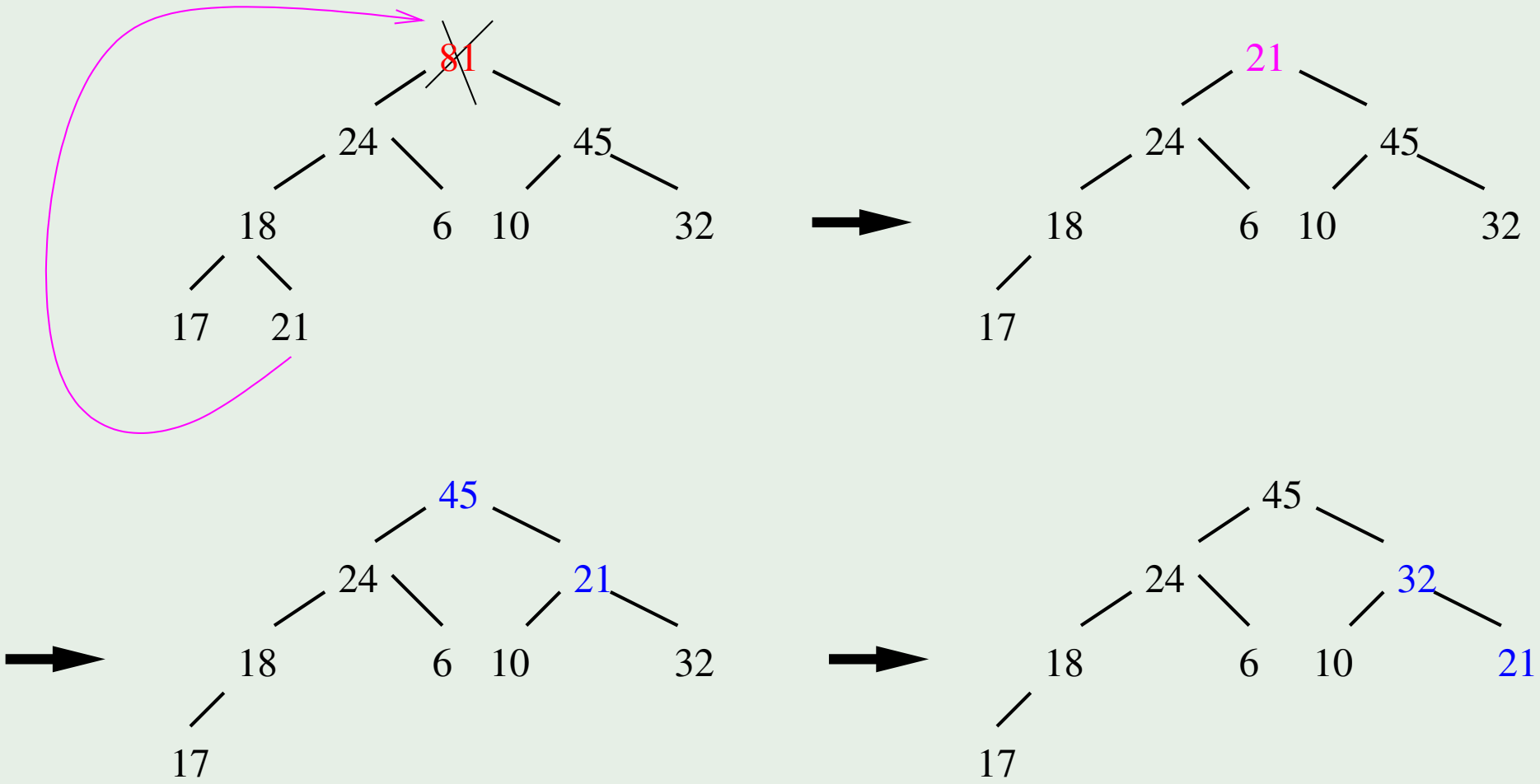
Pour la *suppression* du sommet dans un tas :

1. on supprime la racine et on la remplace par l'élément le plus en bas à droite, ce qui maintient l'arbre complet ;
2. on échange la racine avec le plus grand de ses fils si l'un d'entre eux est plus grand qu'elle, puis à nouveau avec son plus grand fils si l'un d'entre eux est plus grand qu'elle, et ainsi de suite, ce qui maintient l'arbre partiellement ordonné.

Opérations sur les tas

Suppression du sommet

Exemples (Suppression du sommet)



Opérations sur les tas

Suppression du sommet

★★★

Algorithme

```
void heap_remove_max(int *heap, size_t n) {
  heap[0] = heap[n - 1];
  size_t i = 0;
  while (i < (n - 1) / 2) {
    size_t lt = 2 * i + 1;
    size_t rt = 2 * i + 2;
    if (heap[i] > heap[lt] && heap[i] > heap[rt]) {
      break;
    }
    size_t j = (heap[lt] > heap[rt]) ? lt : rt;
    array_swap(heap, i, j);
    i = j;
  }
}
```

Opérations sur les tas

Suppression du sommet

★★

Complexité

L'opération fondamentale est la comparaison ($>$). La fonction `heap_remove_max` parcourt une branche de l'arbre, donc au plus h nœud. Elle fait au maximum deux comparaisons par nœud visité. Donc, la complexité pour un tas de taille n est :

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

Plan

Tri par tas
Principe

★★★

Principe

Le *tri par tas* (*heapsort*) utilise le tableau comme un tas. Il s'effectue en deux étapes :

1. on transforme le tableau en tas par ajout successif des éléments
2. on retire les maximums sucessifs des éléments et on les place à la fin

Tri par tas

Algorithme

★★

Algorithme

```
void array_heap_sort(int *data, size_t n) {
  for (size_t i = 0; i < n; ++i) {
    int value = data[i];
    heap_insert(data, i, value);
  }
  for (size_t i = 0; i < n; ++i) {
    int value = data[0];
    heap_remove(data, n - i);
    data[n - i - 1] = value;
  }
}
```

Tri par tas
Complexité

★★

Complexité

Pour chacune des étapes :

1. on fait n appel à `heap_insert` qui agit sur un tas d'au plus n éléments, pour un coût total de $O(n \log n)$;
2. on fait n appel à `heap_remove_max` qui agit sur un tas d'au plus n éléments, pour un coût total de $O(n \log n)$;

Le tri par tas a donc une complexité de :

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

Le tri par tas est optimal.

Neuvième partie

Graphes

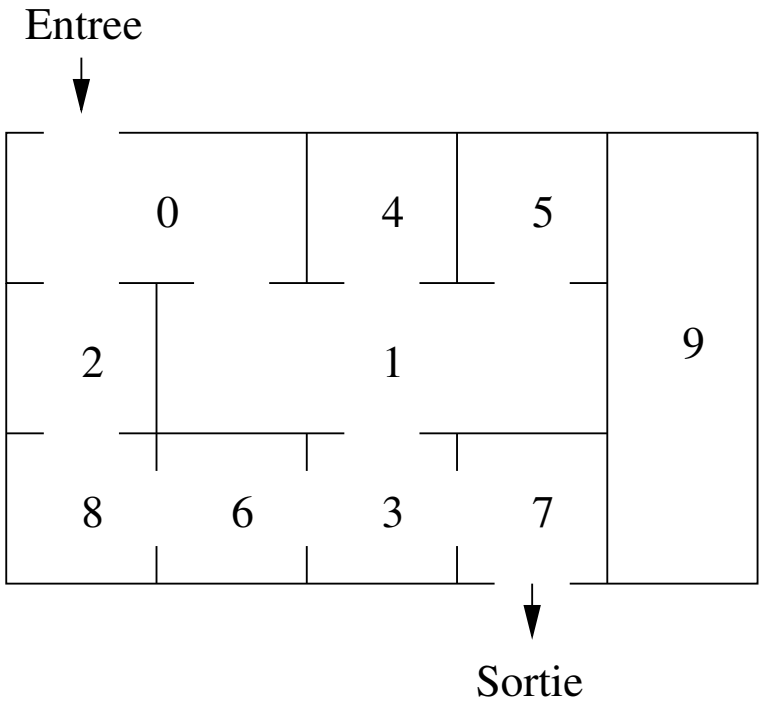
Plan de ce cours

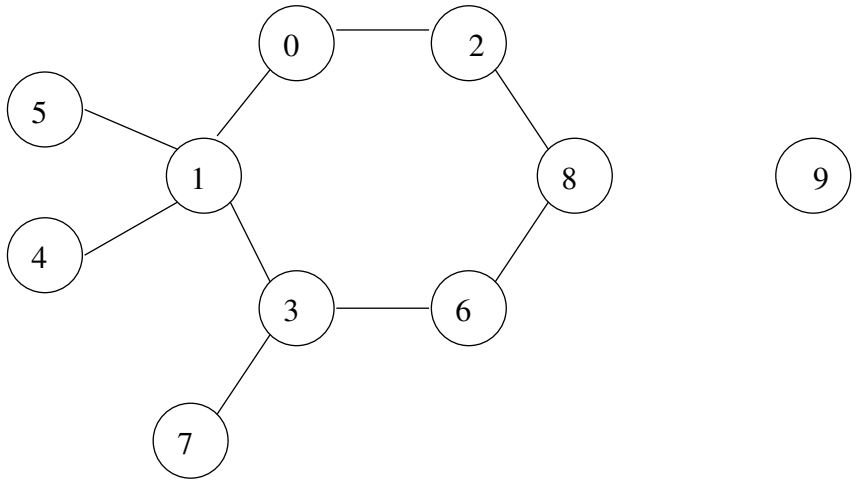
22 Généralités
- Définitions
- Représentations

23 Algorithmes
- Opérations
- Parcours d'un graphe
- Arbre couvrant

Plan

Exemple introductif





Exemple

On peut représenter le labyrinthe (à gauche) à l'aide d'un graphe (à droite)

Graphe, graphe orienté, graphe non-orienté ⋆⋆⋆

Définition (Graphe)

Un *graphe* est un ensemble de points (*vertex*, *vertices*), égalements appelés sommets ou nœuds, reliés par des liens (*edge*, *edges*). Plus précisément, on note le graphe $G = (V, E)$.

Définition (Graphe orienté et non-orienté)

- Un *graphe orienté* est un graphe dans lequel les liens sont unidirectionnels, on les appelle alors des *arcs*.
- Un *graphe non-orienté* est un graphe dans lequel les liens sont bidirectionnels, on les appelle alors *arêtes*.

Extrêmité, successeur, prédécesseur ⋆⋆

Définitions (Extrêmité, degré)

Soit $G = (V, E)$ un graphe non-orienté.

- Soit $e = [u, v] \in E$, u et v sont appelés les *extrêmités* de l'arête e, u et v sont dits *adjacents*
- Soit $u \in V$, on appelle *degré* de u, noté $d(u)$, le nombre d'arêtes ayant pour extrêmité u

Définitions (Sucesseur, prédécesseur, degré entrant, degré sortant)

Soit $G = (V, E)$ un graphe orienté.

- Soit $e = (u, v) \in E$, alors v est appelé le *successeur* de u et u est appelé le *prédécesseur* de v
- Soit $u \in V$, on appelle :
  - *degré entrant* de u, noté $d^-(u)$, le nombre de prédécesseurs de u
  - *degré sortant* de u, noté $d^+(u)$, le nombre de successeurs de u

Degré d'un graphe

Définition (Degré maximum d'un graphe)

Le *degré maximum d'un graphe* G, noté $\Delta(G)$ est le maximum des degrés des sommets du graphe.

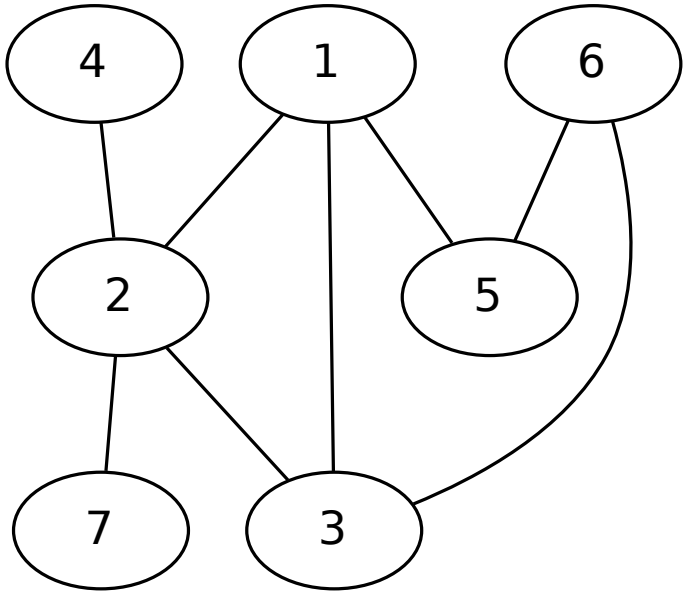
$$\Delta(G) = \max_{u \in V} d(u)$$

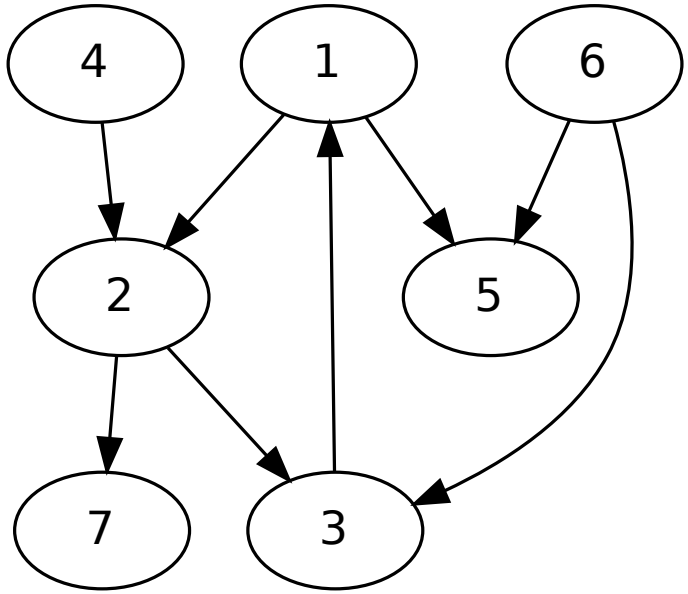
Définition (Degré minimum d'un graphe)

Le *degré minimum d'un graphe* G, noté $\delta(G)$ est le minimum des degrés des sommets du graphes.

$$\delta(G) = \min_{u \in V} d(u)$$

Exemples de graphes





Exemple (Graphe non-orienté)

- $V = \{1, 2, 3, 4, 5, 6, 7\}$
- $E = \{[1, 2], [1, 3], [2, 3], [2, 4], [1, 5], [5, 6], [3, 6], [2, 7]\}$

Exemple (Graphe orienté)

- $V = \{1, 2, 3, 4, 5, 6, 7\}$
- $E = \{(1, 2), (3, 1), (2, 3), (4, 2), (1, 5), (6, 5), (6, 3), (2, 7)\}$

Graphe simple ⋆⋆

Définition (Boucle)

Une *boucle* est une arête (ou un arc) ayant le même sommet comme extrêmités.

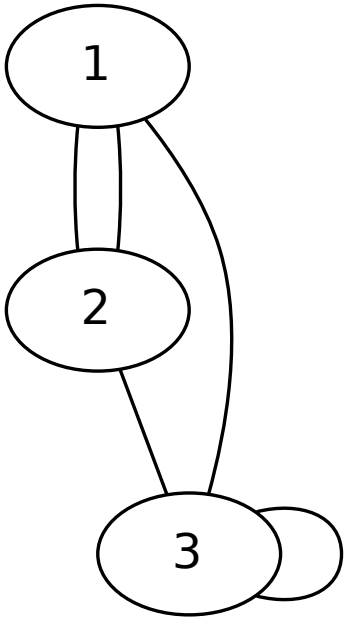
Définition (Arête multiple)

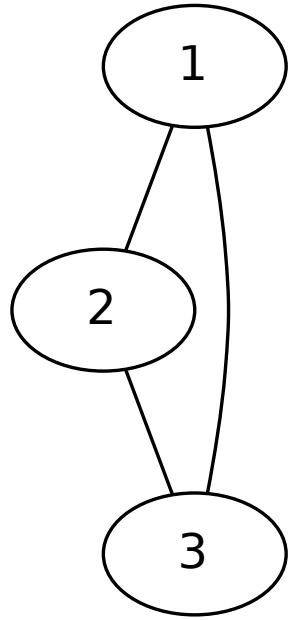
Une *arête multiple* est un ensemble d'arêtes parallèles, c'est-à-dire d'arêtes qui partagent les mêmes extrêmités.

Définition (Graphe simple)

Un *graphe simple* est un graphe sans boucle ni arête multiple.

Exemples de graphes





Exemple (Graphe non-simple)

- Arête multiple entre les sommets 1 et 2
- Boucle autour du sommet 3

Exemple (Graphe simple)

- Le graphe est simple : ni arête multiple, ni boucle

Chaîne, cycle, chemin, circuit ⋆⋆

Définition (Chaîne et cycle)

Soit $G = (V, E)$ un graphe non-orienté.

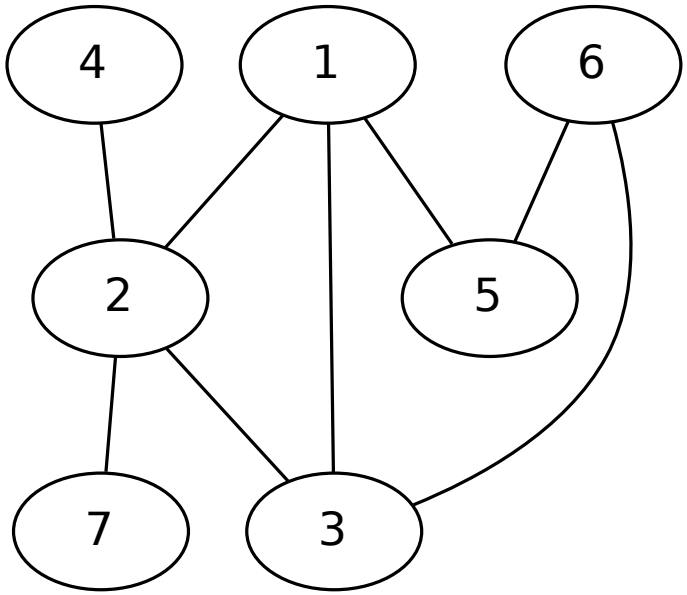
- Une *chaîne* de longueur l est une suite $(u_0, u_1, \ldots, u_l)$ de $l + 1$ sommets tels que pour tout $k \in [0, l[$, (u_k, u_{k+1}) est une arête de G.
- Un *cycle* est une chaîne dont les sommets de départ et de fin sont identiques ($u_0 = u_l$).

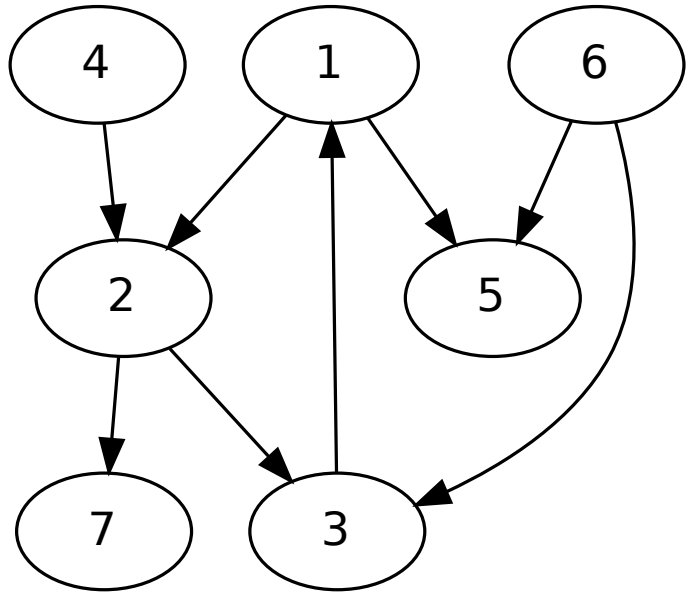
Définition (Chemin et circuit)

Soit $G = (V, E)$ un graphe orienté.

- Un *chemin* de longueur l est une suite $(u_0, u_1, \ldots, u_l)$ de $l + 1$ sommets tels que pour tout $k \in [0, l[$, (u_k, u_{k+1}) est un arc de G.
- Un *circuit* est un chemin dont les sommets de départ et de fin sont identiques ($u_0 = u_l$).

Exemples de graphes





Exemple (Chaîne et cycle)

- $(5, 1, 3, 2, 4)$ est un chaîne de longueur 4 entre 5 et 4
- $(3, 1, 5, 6, 3)$ est un cycle

Exemple (Chemin et circuit)

- $(6, 3, 1, 2, 7)$ est un chemin de longueur 4 entre 6 et 7
- $(1, 2, 3, 1)$ est un circuit

Connexité d'un graphe ⋆⋆

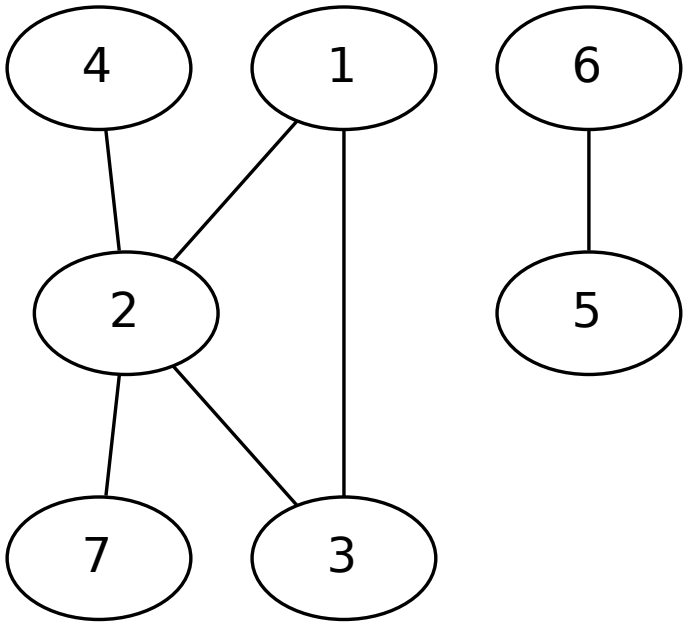
Définition (Graphe connexe)

Un graphe non-orienté G est *connexe* si pour toute paire de sommets distincts (u, v), il existe une chaîne de u à v.

Définition (Graphe fortement connexe)

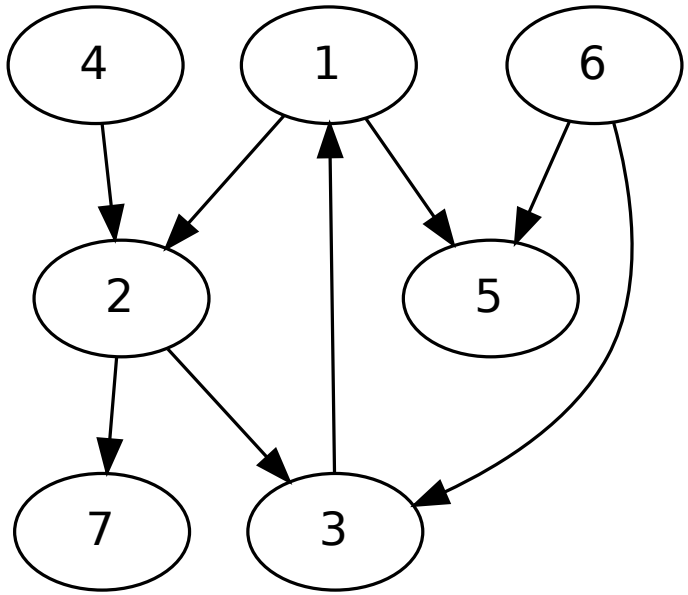
Un graphe orienté G est *fortement connexe* si pour toute paire de sommets distincts (u, v), il existe une chemin de u à v et un chemin de v à u.

Exemples de graphes



Exemple (Connexité)

- Le graphe n'est pas connexe



Exemple (Connexité)

- Le graphe est connexe
- Le graphe n'est pas fortement connexe

Graphe acyclique ou sans circuit ⋆⋆

Définition (DAG)

Un *DAG* (*Directed Acyclic Graph*) est un graphe connexe orienté sans circuit.

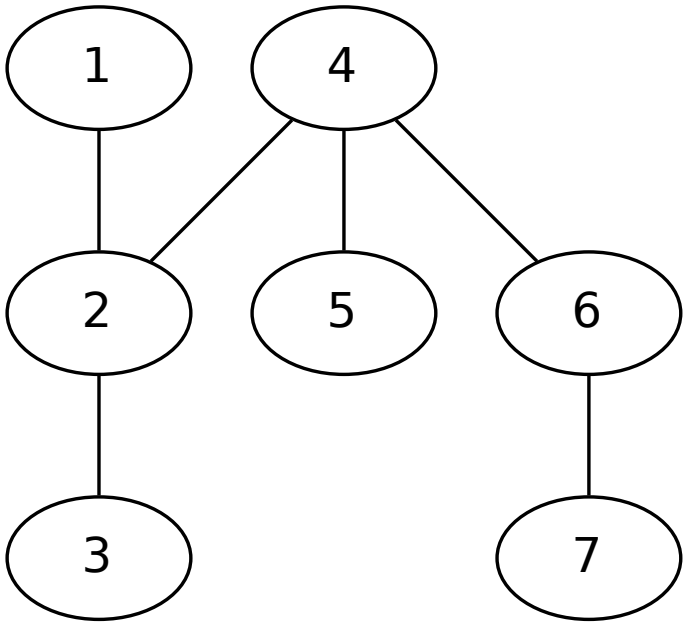
Définition (Arbre)

Un *arbre* est un graphe connexe non-orienté acyclique.

Remarque

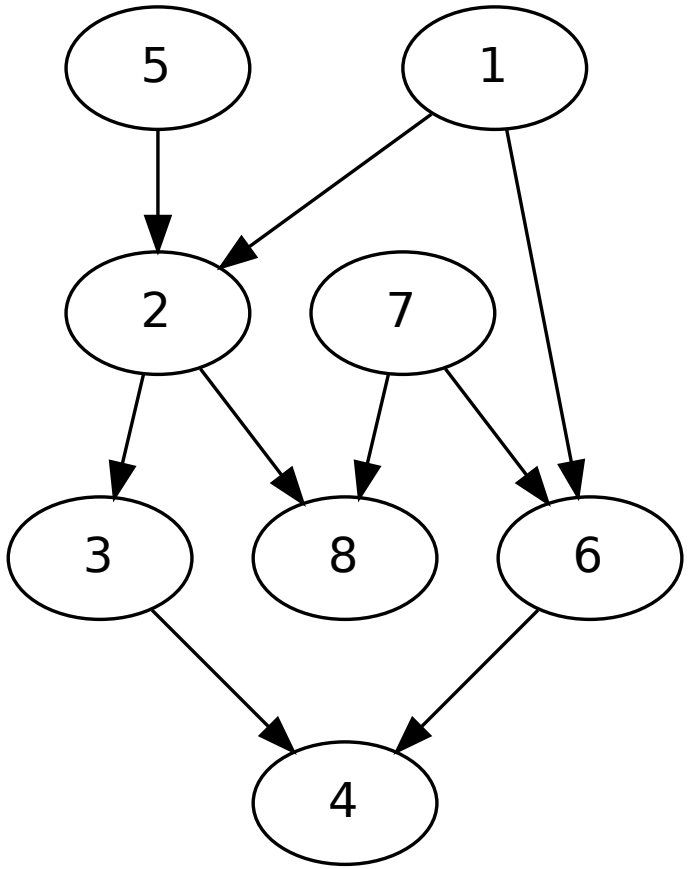
Si on choisit un sommet r quelconque dans un arbre, il est possible d'enraciner l'arbre en r, c'est-à-dire orienter toutes les arêtes de sorte qu'il existe un chemin de r à tous les autres nœuds. On obtient alors un *arbre enraciné* ou *arborescence*.

Exemples de graphes



Exemple (Arbre)

- Le graphe est un arbre



Exemple (DAG)

- Le graphe est un DAG

Graphe complet, densité ⋆⋆

Définition (Graphe complet)

Le *graphe complet* à n sommet, noté K_n, est le graphe dans lequel chaque sommet est relié à tous les autres sommets par une arête.

Lemme (Relation entre nombre de sommets et nombre d'arêtes)

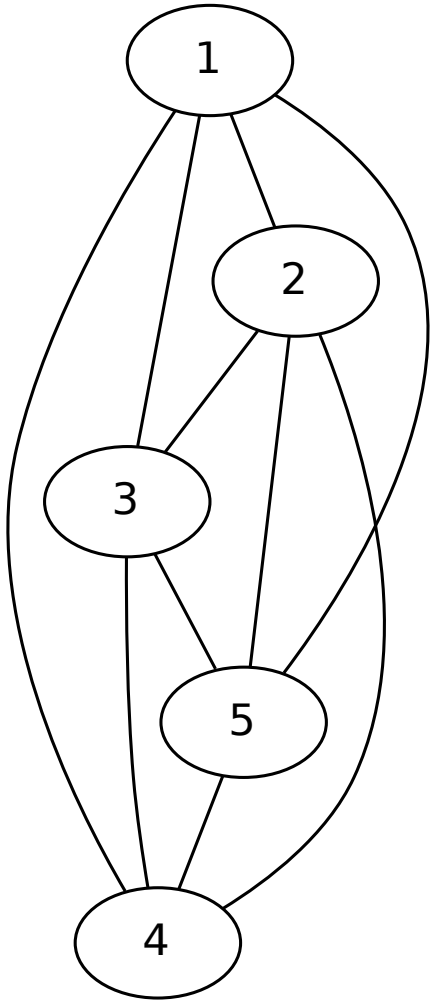
Pour un graphe simple connexe avec n sommets et m arêtes :

$$n - 1 \leq m \leq \frac{n(n-1)}{2}$$

Définition (Densité)

La densité d'un graphe avec n sommets et m arêtes est le rapport entre le nombre d'arêtes et le nombre d'arêtes possibles. C'est-à-dire : $\frac{2m}{n(n-1)}$

Exemples de graphes



Exemple (Graphe complet)

- K_5

Graphe valué ⋆⋆

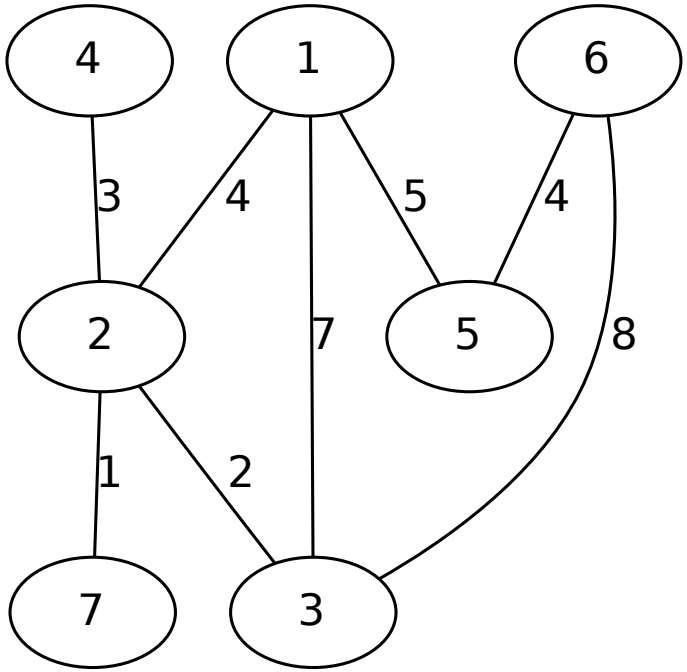
Définition (Graphe valué)

Un *graphe valué* ou graphe pondéré, est un graphe muni d'une fonction de valuation (ou fonction de coût) qui associe une valeur (ou coût ou poids) à chaque arête.

Remarque

Il est aussi possible de valuer les sommets.

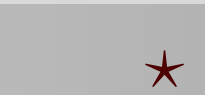
Exemples de graphes



Exemple (Graphe valué)

- Le graphe est valué

Les graphes sont partout



Que peut-on modéliser avec un graphe ?

- Carte routière
- Relation sur un réseau social
- Réseau informatique
- Dépendances entre cibles dans un Makefile
- Automate à états fini
- ...

Plan

Représentation d'un graphe

Généralités

★★

Représentation d'un graphe

Il existe plusieurs manières de représenter un graphe :

- liste d'adjacence
- matrice d'adjacence
- matrice d'incidence

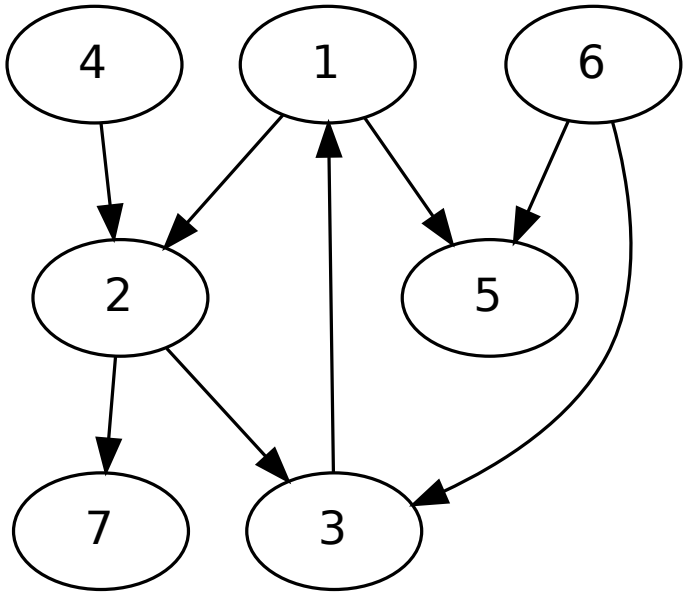
Représentation d'un graphe

Liste d'adjacence

★★

Liste d'adjacence

Pour un graphe $G = (V, E)$, la *liste d'adjacence* est un tableau A représentant l'ensemble des sommets, et où chaque case i du tableau donne la liste des sommets adjacents au sommet u_i.



Exemple (Liste d'adjacence)

- $A[1] = \{2, 5\}$
- $A[2] = \{3, 7\}$
- $A[3] = \{1\}$
- $A[4] = \{2\}$
- $A[5] = \varnothing$
- $A[6] = \{3, 5\}$
- $A[7] = \varnothing$

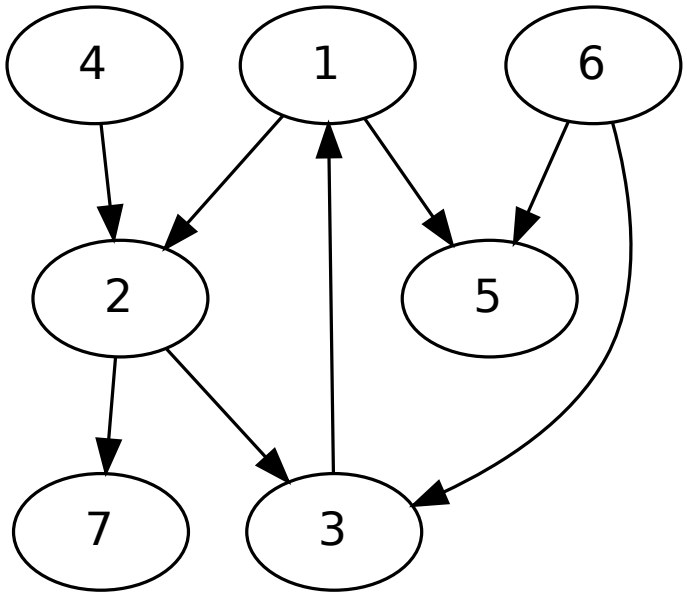
Représentation d'un graphe

Matrice d'adjacence

★★

Matrice d'adjacence

Pour un graphe $G = (V, E)$, la *matrice d'adjacence* M est une matrice de taille $|V| \times |V|$ telle que $M_{ij} = 1$ s'il existe un arc de u_i à u_j, c'est-à-dire si $(u_i, u_j) \in E$, et 0 sinon.



Exemple (Matrice d'adjacence)

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

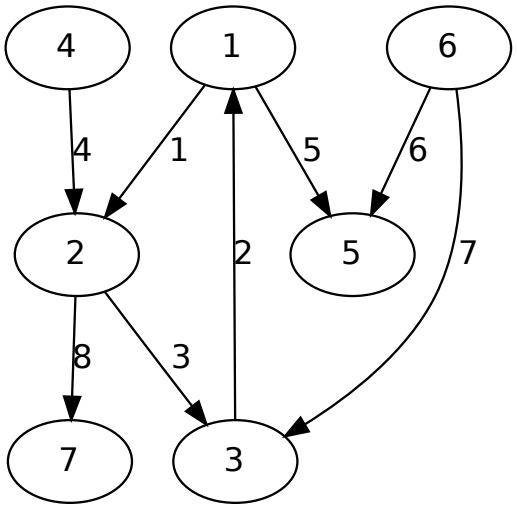
Représentation d'un graphe

Matrice d'incidence

★★

Matrice d'incidence

Pour un graphe $G = (V, E)$, la *matrice d'incidence* M est une matrice de taille $|V| \times |E|$ telle que $M_{ij} = -1$ si l'arc e_j sort du sommet u_i, $M_{ij} = 1$ si l'arc e_j entre dans le sommet u_i, 0 sinon.



Exemple (Matrice d'incidence)

$$M = \begin{pmatrix} -1 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 1 & 0 & 0 & 0 & -1 \\ 0 & -1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Plan

Opérations élémentaires sur un graphe ⋆⋆

Opérations élémentaires

- sommets adjacents, noté `adjacent`, défini par :

$$\texttt{adjacent}(G, u) = \{v \in V, [u, v] \in E\}$$

- sommets successeurs, noté `succ`, défini par :

$$\texttt{succ}(G, u) = \{v \in V, (u, v) \in E\}$$

- sommets prédécesseurs, noté `pred`, défini par :

$$\texttt{pred}(G, u) = \{v \in V, (v, u) \in E\}$$

Plan

Parcours d'un graphe

Définition

★★

Définition (Parcours)

Un *parcours* d'un graphe est la visite successive de tous les sommets dans un certain ordre et au cours duquel on effectue une opération. On distingue deux types de parcours :

- parcours en profondeur
- parcours en largeur

Remarques

- Contrairement aux arbres, un parcours de graphe nécessite de pouvoir marquer les sommets pour savoir s'ils ont déjà été visité ou pas, de manière à ne pas tourner en rond.
- Contrairement aux arbres, il n'y a pas un sommet particulier par lequel on commence le parcours, c'est pourquoi on indique toujours le sommet de départ du parcours

Parcours d'un graphe

Parcours en profondeur

★★

Parcours en profondeur

Le *parcours en profondeur* (*depth-first search*, ou DFS) d'un graphe permet de parcourir le graphe en privilégiant les sommets éloignés du départ. On parcoure le graphe de manière récursive.

Remarque

L'ordre dans lequel on parcoure les successeurs d'un sommet n'est pas défini a priori et va conditionner le parcours global.

Parcours d'un graphe

Parcours en profondeur

★★

Parcours en profondeur

```
function DepthFirstSearch(G, s)
    Mark(s)
    for u in adjacent(G, s) do
        if not IsMarked(u) then
            DepthFirstSearch(G, u)
        end if
    end for
end function
```

Parcours d'un graphe

Parcours en largeur

★★

Parcours en largeur

Le *parcours en largeur* (*breadth-first search*, ou BFS) d'un graphe permet de parcourir le graphe en privilégiant les sommets proche du départ. On parcoure le graphe de manière itérative.

Remarque

L'ordre dans lequel on parcoure les successeurs d'un sommet n'est pas défini a priori et va conditionner le parcours global.

Parcours d'un graphe

Parcours en largeur

★★

Parcours en largeur

```
function BreadthFirstSearch(G, s)
    q ← queue()
    Mark(s) ; enqueue(q, s)
    while not empty(q) do
        u ← peek(q)
        for v in adjacent(G, u) do
            if not IsMarked(v) then
                Mark(v) ; enqueue(q, v)
            end if
        end for
        dequeue(q)
    end while
end function
```

Plan

Arbre couvrant ⋆⋆

Définition (Arbre couvrant)

Un arbre couvrant (*spanning tree*) A d'un graphe G est un graphe qui a les mêmes sommets que G, et dont les arêtes sont un sous-ensemble des arêtes de G et qui est un arbre, c'est-à-dire un graphe connexe sans cycle.

Remarques

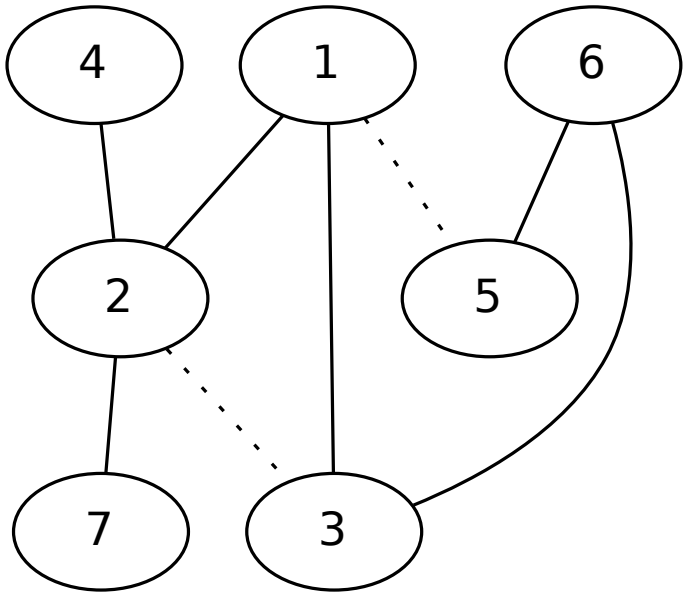
- On va voir ici comment construire un arbre couvrant dans le cas d'un graphe G non valué. Il existe des algorithmes qui permettent, pour un graphe G valué de contruire des arbres couvrants de poids minimal (*Minimum Spanning Tree*).
- Les arbres couvrants sont utilisés notamment en réseau pour communiquer sur des réseaux locaux en évitant les boucles, grâce au Spanning Tree Protocol (STP).

Arbre couvrant
Principe

★★

Arbre couvrant

L'idée pour construire un arbre couvrant est de parcourir le graphe en profondeur à partir d'un nœud r et de construire un tableau p dans lequel $p[i]$ est le père de i dans l'arbre.



Exemple (Arbre couvrant)

i	$p[i]$
4	-
2	4
1	2
3	1
6	3
5	6
7	2

Arbre couvrant

Algorithme

★★

Arbre couvrant

```
function SpanningTree(G, s, p)
    Mark(s)
    for u in adjacent(G, s) do
        if not IsMarked(u) then
            p[u] ← s
            SpanningTree(G, u, p)
        end if
    end for
end function
```

Dixième partie

Plus court chemin dans un graphe

Plan de ce cours

Plan

Plus court chemin dans un graphe ⋆⋆

Définition

Définition du problème

Problème *Plus court chemin dans un graphe*

Données : un graphe $G = (V, E)$ valué, un sommet initial s, et un sommet final f

Résultat : le plus court chemin depuis s jusqu'à f

Cette version du problème s'appelle *plus court chemin entre deux sommets* (*single-pair shortest path problem*). Il existe des variantes :

- plus court chemin depuis une source (*single-source shortest path problem*) où on doit trouver le plus court chemin depuis une source vers tous les sommets du graphe
- plus court chemin entre tous les sommets (*all-pairs shortest path problem*) où on doit trouver le plus court chemin entre toutes les paires de sommets du graphe

Plus court chemin dans un graphe

Court ?

★★

Que signifie «court» ?

- Le graphe $G = (V, E)$ est valué, c'est-à-dire qu'il existe une fonction $w : E \to \mathbb{R}$ qui donne la valuation de chaque arête. On l'appelle aussi le poids (*weight*).
- Soit $P = (u_1, u_2, \ldots, u_k)$ un chemin dans G, on peut définir $w(P)$ par :

$$w(P) = \sum_{i=1}^{k-1} w(u_i, u_{i+1})$$

- Soit $\mathcal{P}_s^f$ l'ensemble des chemins entre les sommets s et f, le plus court chemin P^* entre s et f est défini par :

$$P^* = \arg \min_{P \in \mathcal{P}_s^f} w(P)$$

Plus court chemin dans un graphe

Considérations pratiques

Considérations pratiques

Nous allons voir deux algorithmes de plus court chemin (depuis une source) :

- l'algorithme de Bellman-Ford
- l'algorithme de Dijkstra

En pratique, ces algorithmes ne vont pas renvoyer un seul chemin mais tous les chemins depuis la source. De plus, on a souvent besoin de connaître les distances entre les sommets et la source. Donc, chaque algorithme calculera :

- un tableau π qui donne pour chaque sommet son prédecesseur dans le plus court chemin
- un tableau d qui donne pour chaque sommet sa distance depuis la source

Plus court chemin dans un graphe

Initialisation des résultats

Initialisation des résultats

function InitResults(d, π)
    for $v \in V$ **do**
        $d[v] \leftarrow +\infty$
        $\pi[v] \leftarrow \varnothing$
    end for
end function

Plan

Algorithme de Bellman-Ford
Généralités

Généralités

- L'algorithme de Bellman-Ford a été inventé en 1956 par Lester Ford Jr, puis en 1958 par Richard Bellman.
- Il permet de trouver un plus court chemin depuis une source.
- Son grand avantage est qu'il s'applique à des graphes valué quelconque, et qu'il est capable de détecter des cycles négatifs.

Algorithme de Bellman-Ford

Algorithme

★★

Algorithme

function BellmanFord(G, s)
    InitResults(d, π)
    $d[s] \leftarrow 0$
    for i **from** 1 **to** $|V| - 1$ **do**
        for $(u, v) \in E$ **do**
            if $d[v] > d[u] + w(u, v)$ **then**
                $d[v] \leftarrow d[u] + w(u, v)$; $\pi[v] \leftarrow u$
            end if
        end for
    end for
    return HasCycle(G, d)
end function

Algorithme de Bellman-Ford

Détection de cycle

★★

Détection de cycle

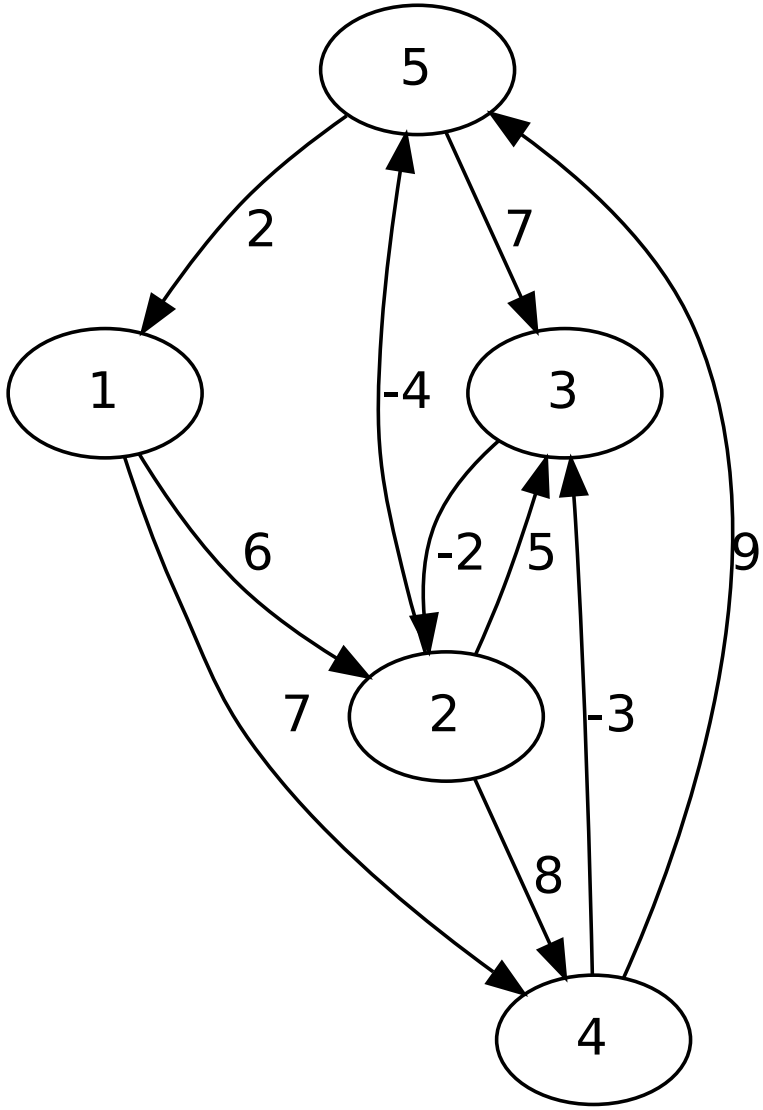
function HasCycle(G, d)
    for $(u, v) \in E$ **do**
        if $d[v] > d[u] + w(u, v)$ **then**
            return `true`
        end if
    end for
    return `false`
end function

Remarque

Dans le cas où il n'y a pas de cycle, on peut arrêter la boucle principale s'il n'y a pas eu de mise à jour du tableau d au cours de la boucle précédente.

Algorithme de Bellman-Ford

Exemple



Plan

Algorithme de Bellman-Ford

Preuve

Idée de la preuve de l'algorithme

On prouve l'algorithme en montrant par récurrence qu'après la i^{e} boucle :

1. si $d[u]$ n'est pas $+\infty$, c'est la distance d'un chemin de s à u ;
2. s'il existe un chemin de s à u d'au plus i arêtes, alors $d[u]$ est au plus la distance du plus court chemin de s à u avec au plus i arêtes.

Algorithme de Bellman-Ford
Complexité

★★

Complexité

Pour un graphe $G = (V, E)$, l'algorithme effectue :

- $O(|V|)$ opérations pour l'initialisation
- $O(|V| \times |E|)$ opérations pour la partie principale
- $O(|E|)$ opérations pour la détection de cycle

Donc, la complexité de l'algorithme de Bellman-Ford est :

$$O(|V| \times |E|)$$

Pour un graphe dense, la complexité de Bellman-Ford est :

$$O(|V|^3)$$

Plan

Algorithme de Dijkstra
Généralités

Généralités

- L'algorithme de Dijkstra a été inventé en 1956, et publié en 1959 par Edsger Dijkstra.
- Il permet de trouver un plus court chemin depuis une source.
- Il a une meilleure complexité que l'algorithme de Bellman-Ford mais il ne s'applique qu'au graphe valué avec des valeurs positives.

Algorithme de Dijkstra

Algorithme

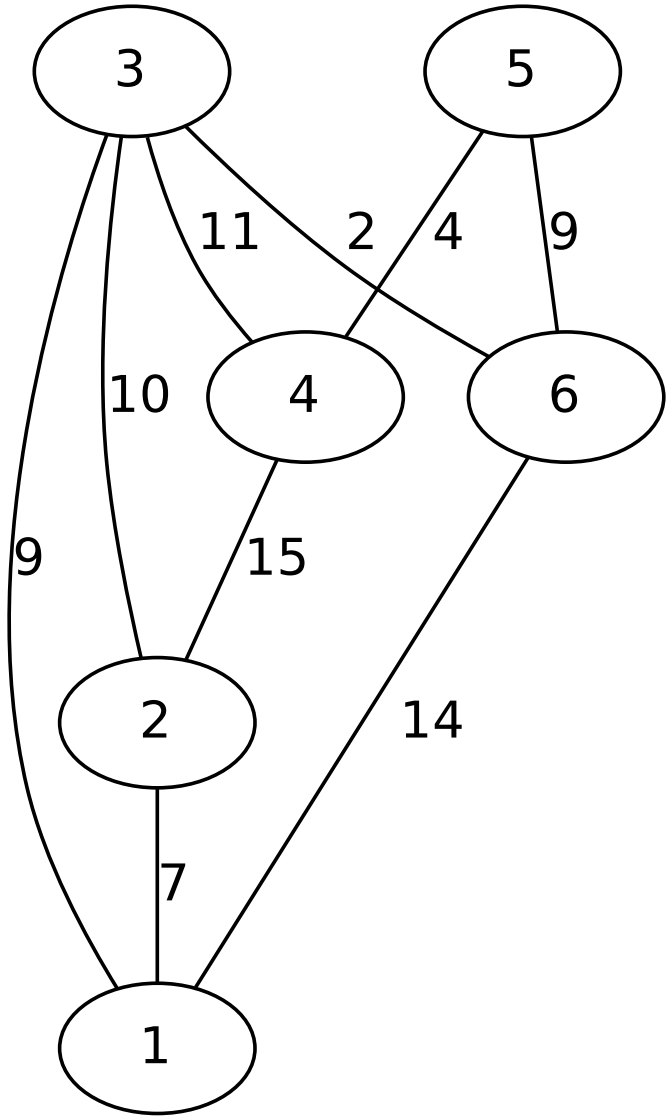
★★

Algorithme

function DIJKSTRA(G, s)
    INITRESULTS(d, π)
    $d[s] \leftarrow 0$; INIT(q, V)
    while not `empty`(q) **do**
        $u \leftarrow$ EXTRACTMIN(q)
        for $v \in$ `succ`(u) **do**
            if $d[v] > d[u] + w(u, v)$ **then**
                $d[v] \leftarrow d[u] + w(u, v)$; $\pi[v] \leftarrow u$
                DECREASEKEY(q, v)
            end if
        end for
    end while
end function

Algorithme de Dijkstra

Exemple



Plan

Algorithme de Dijkstra
Complexité

★★

Complexité

Pour un graphe $G = (V, E)$, l'algorithme parcoure chaque sommet une fois dans la boucle principale et parcoure au maximum une fois chaque arête dans la condition. Donc, l'algorithme effectue :

- $O(|V|)$ appels à ExtractMin
- $O(|E|)$ appels à DecreaseKey

Tout dépend donc de la manière dont on implémente la structure q. Il est possible d'utiliser :

- une liste chaînée
- un tas

Algorithme de Dijkstra
Complexité avec une liste chaînée

★★

Complexité avec une liste chaînée

Avec une liste chaînée :

- la fonction ExtractMin a une complexité en $O(|V|)$
- la fonction DecreaseKey a une complexité en $O(1)$

Donc, la complexité de l'algorithme de Dijkstra est :

$$O(|V|^2 + |E|) = O(|V|^2)$$

Algorithme de Dijkstra

Complexité avec un tas

★★

Complexité avec un tas

Avec un tas :

- la fonction ExtractMin a une complexité en $O(\log |V|)$
- la fonction DecreaseKey a une complexité en $O(\log |V|)$

Donc, la complexité de l'algorithme de Dijkstra est :

$$O((|V| + |E|) \times \log |V|)$$

Remarque

Il existe des tas dont l'implémentation permet d'avoir une fonction DecreaseKey en $O(1)$ amorti, ce qui amène à une complexité de $O(|V| \log |V| + |E|)$.

Onzième partie

Le langage C++

Plan de ce cours

27 Le langage C++
- Généralités
- Éléments du langage
- Éléments de la bibliothèque standard

Plan

Historique du C++

Historique du C++

- 1979 : Création du «C with Classes», par Bjarne Stroustrup
- 1983 : «C with Classes» devient C++
- 1998 : Normalisation par l'ISO, C++98
- 2003 : Mise à jour de la norme : C++03
- 2007 : Ajout d'un rapport technique : C++ TR1
- 2011 : Mise à jour de la norme : C++11
- 2014 : Mise à jour prévue de la norme : C++14
- 2017 : Mise à jour prévue de la norme : C++17

Langage C++

Langage C++

- Compatible avec C (largement)
- Programmation procédurale
- Programmation orienté objet
- Programmation générique
- Et bien plus !

Quelques pointeurs utiles

Quelques pointeurs utiles

- Le langage C++, Henri Garreta
  `http://henri.garreta.perso.luminy.univmed.fr/`
- C++ Reference
  `http://en.cppreference.com/w/cpp`
- C++ FAQ
  `http://isocpp.org/faq`

Plan

Éléments bas niveau du langage

Éléments bas niveau du langage

- Type `bool`
- Opérateurs `new` et `delete`

```
int *ptr = new int;
delete ptr;
```

- Valeurs par défaut pour les paramètres de fonctions

```
int myfunc(int a, float b = 3.14, bool c = false);
```

- Fonctions `inline`

```
inline int abs(int x) { return x > 0 ? x : -x; }
```

Espaces de noms

Espaces de noms

- Possibilité de définir des espaces de nom via `namespace`

```
namespace foo { struct bar { int baz; }; }
foo::bar var;
```

- La bibliothèque standard est dans l'espace de nom `std`

Remarque

- Ne pas utiliser `using namespace std;` comme on le voit dans beaucoup de tutoriels !
- Les espaces de noms ont une utilité !

Références

Références

- Pour tout type `T`, il est possible de définir une référence sur `T`, noté `T&`
- Une référence est une sorte de pointeur non-nul, mais s'utilise comme une variable normale

```
int i = 1;
int& j = i;
j = 2;
```

- Utilisés dans les paramètres d'une fonction ou pour le type de retour

```
void do(const BigType& obj);
int& get(const char *name);
get("toto") = 3;
```

Classes
Déclaration

Déclaration d'une classe

```
class Foo {
public:
  Foo(); // constructeur
  Foo(const Foo&); // constructeur par copie
  ~Foo(); // destructeur

  Foo& operator=(const Foo&); // affectation

  void public_method();
  void const_method() const;
private:
  void private_method();
};
```

Classes
Déclaration

Remarques

- `class` et `struct` sont synonymes. Seule différence : la visibilité par défaut est `public` pour `struct` et `private` pour `class`.
- Les méthodes ne sont pas polymorphes par défaut, nécessité de mettre le mot-clef `virtual`

```
class Bar {
public:
  void method(); // normale
  virtual void virtual_method(); // virtuelle
  virtual void pure_method() = 0; // virtuelle pure
}
```

Classes
Déclaration

Héritage

- Héritage simple

```
class Baz : public Bar {
public:
  void method();
  virtual void virtual_method();
  virtual void pure_method();
}
```

- Héritage multiple (à éviter)

```
class Qux : public Baz, public Foo {
}
```

Classes
Définition

Définition

```
Foo::Foo() { }

Foo::~Foo() { }

void Foo::public_method() { }

void Foo::const_method() const { }

void Foo::private_method() { }
```

Classes
Initialisation

Initialisation

```
class Toto {
public:
  Toto(int data);
private:
  int m_data;
}

Toto:Toto(int data) {
  m_data = data; // NON !
}

Toto:Toto(int data) : m_data(data) // OUI !
{ }
```

Classes

this

this

- Dans une méthode d'une classe `C`, `this` a le type `C*` pour les méthodes non-`const` et `const C*` pour les méthodes `const`
- Rarement utilisé

Templates

Templates

- Les templates permettent une programmation générique
- Template de fonctions

```
template<typename T>
T max(T a, T b) {
  if (a < b) {
    return b;
  }
  return a;
}
```

Templates

Templates

- Template de classes

```
template<typename T>
class Gruik {
  Gruik();
private:
  T m_data;
}
```

Exceptions

Exceptions

- Il existe des exceptions en C++
- `throw` pour envoyer une exception
- `try { ... } catch (...) { ... }`
- Pas de déclaration systématique pour les fonctions
- Des exceptions standards
- À éviter au maximum !

Run-time Type Information (RTTI)

Run-time Type Information (RTTI)

- Il est possible d'accéder à des informations sur les classes à l'exécution
- `typeid(expr)`
- Ajoute un surplus en terme de mémoire
- Peut être désactivé à la compilation avec `-fno-rtti`

Transtypage

Transtypage

- Plus sûr qu'en C
- `static_cast<T>(expr)` : à peu près l'équivalent du transtypage C `(T) expr`
- `dynamic_cast<T>(expr)` : transtype des types de classes filles avec vérification à l'exécution (en utilisant le RTTI)
- `const_cast<T>(expr)` : pour changer l'attribut `const` d'une expression
- `reinterpret_cast<T>(expr)` : pour des choses dangereuses (pointeur vers entier, etc)

Plan

<string>

<string>

- Un vrai type chaîne de caractère : `std::string`
- Avec toutes les opérations attendues :
  - copie
  - sous-chaîne
  - recherche
  - concaténation
- En fait, `std::basic_string<char>`

<iostream>

<iostream>

- Bibliothèque d'entrée/sortie à base de flux
- Types disponibles :
  - `std::ostream` : flux en écriture, en particulier `std::ofstream`
  - `std::istream` : flux en lecture, en particulier `std::ifstream`
- Fondée sur les opérateurs d'extraction (>>) et d'envoi (<<) dans le flux, définis pour tous les types de base et définissable pour les types utilisateurs

  ```
  std::ostream& operator<<(std::ostream&, const Foo&);
  ```
- `std::cin`, `std::cout`, `std::cerr` : flux standard

<iostream>

Hello World

```
#include <iostream>
int main() {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Hello World

```
#include <iostream>
int main() {
  std::cout << "Alice a " << 20 << " ans";
  std::cout << " et mesure " << 1.70 << "m" << std::endl;
  return 0;
}
```

Standard Template Library

Standard Template Library (STL)

- Bibliothèque avec
  - des conteneurs
  - des itérateurs
  - des algorithmes génériques
- Conçue par Alexander Stepanov en 1992–1993
- Fondée sur les templates
- Une brique de base essentielle à utiliser !

Standard Template Library
Conteneurs

Conteneurs séquentiels

- `std::vector<T>` dans `<vector>`
- `std::list<T>` dans `<list>`
- `std::deque<T>` dans `<deque>`

Conteneurs associatifs

- `std::set<T>` et `std::multiset<T>` dans `<set>`
- `std::map<K,V>` et `std::multimap<K,V>` dans `<map>`

Adaptateurs de conteneurs

- `std::stack<T>` dans `<stack>`
- `std::queue<T>` dans `<queue>`
- `std::priority_queue<T>` dans `<queue>`

Standard Template Library
Itérateurs

Itérateurs

- Un itérateur est une abstration d'un pointeur
- Chaque conteneur `C<T>` définit un type itérateur `C<T>::iterator`
- Chaque conteneur a des méthodes `begin()` et `end()` qui renvoie respectivement un itérateur sur le début et la fin du conteneur
- Permet de parcourir les données d'un conteneur
  `for (C<T>::iterator it = c.begin(); it != c.end(); ++it)`
- L'accès à la donnée se fait par l'opérateur `*` (`*it`) ou par l'opérateur `->` (`it->field`) pour l'accès aux champs ou aux méthodes de la donnée de type `T`

Standard Template Library
Algorithmes génériques

Algorithmes génériques

- Algorithmes avec template prenant en paramètre des itérateurs
- Non-modifiant : `std::for_each`, `std::count`, `std::find`, etc.
- Modifiant : `std::copy`, `std::transform`, `std::reverse`, `std::unique`, etc.
- Partition et tri : `std::partition`, `std::sort`
- Recherche dichotomique : `std::lower_bound`, `std::upper_bound`, `std::binary_search`
- Ensemble ordonné : `std::merge`, `std::includes`, etc.
- Numérique : `std::accumulate`, `std::inner_product`, `std::partial_sum`

Bibliothèque standard du C

Bibliothèque standard du C

- Accès aux fonctions de la bibliothèque standard du C
- En-tête spéciaux : `<cstdio>`, `<cstdlib>`, `<cmath>`, `<cassert>`
- Plongé dans l'espace de nom `std`. Exemple : `std::printf`

C'est tout pour le moment...

Des questions ?