

# INF2010 – ASD

## Monceaux

# Plan

- Définition de monceau et implémentation
- Insertion et retrait d'éléments
- Construction d'un monceux
- Tri par monceau

# Plan

- Définition de monceau et implémentation
- Insertion et retrait d'éléments
- Construction d'un monceux
- Tri par monceau

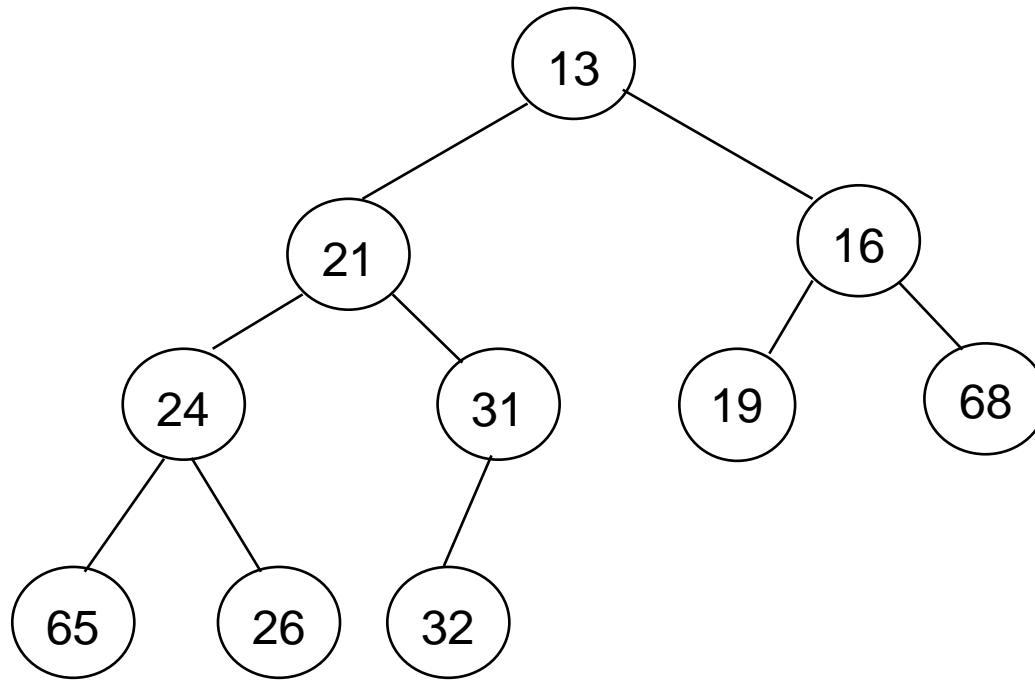
# Monceau - définition

- Un arbre complet tel que pour chaque noeud N dont le parent est P, la clé de P est plus petite que celle de N
- Normalement implémenté dans un tableau
- Insertion et retrait sont  $O(\lg N)$  dans le pire cas
- Recherche du plus petit élément est  $O(1)$  dans le pire cas

# Monceau – implémentation

- On utilise un tableau pour implémenter l'arbre
- On met la racine de l'arbre à la position 1, plutôt que la position 0, ce qui facilite les opérations
- Pour chaque nœud  $i$ , les indices sont donc  $2*i$  pour le fils gauche et  $2*i + 1$  pour le fils droit

# Monceau - exemple

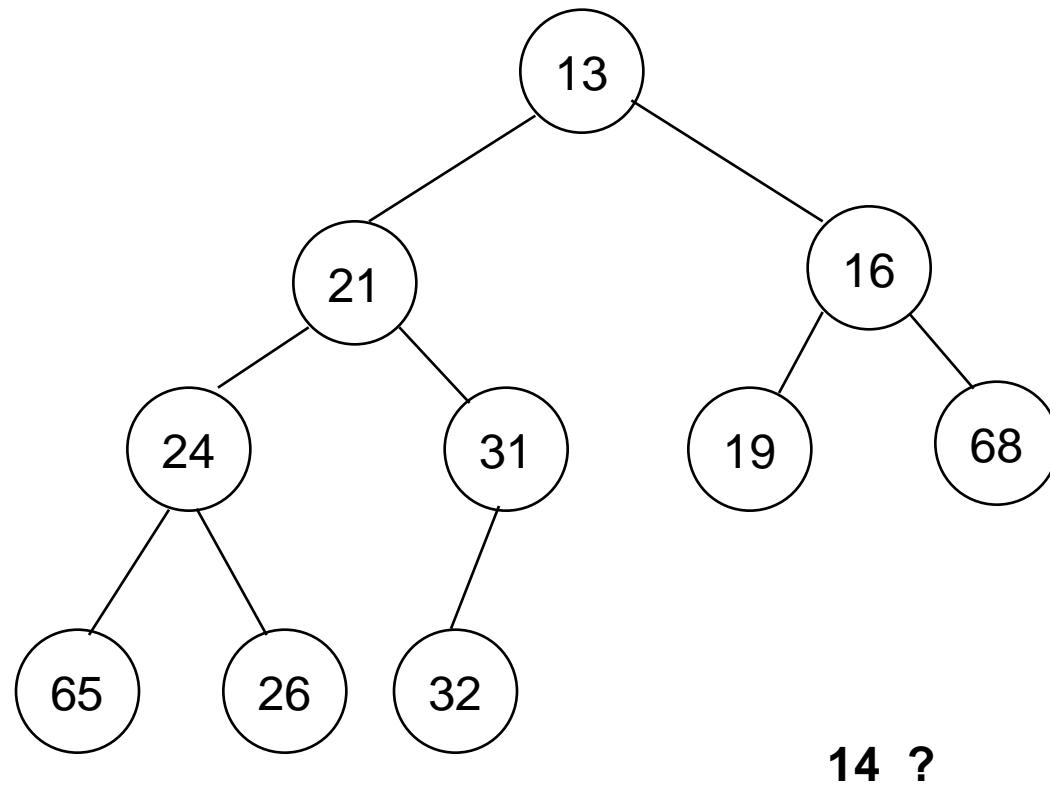


0	1	2	3	4	5	6	7	8	9	10	11
	13	21	16	24	31	19	68	65	26	32	

# Plan

- Définition de monceau et implémentation
- Insertion et retrait d'éléments
- Construction d'un monceau
- Tri par monceau

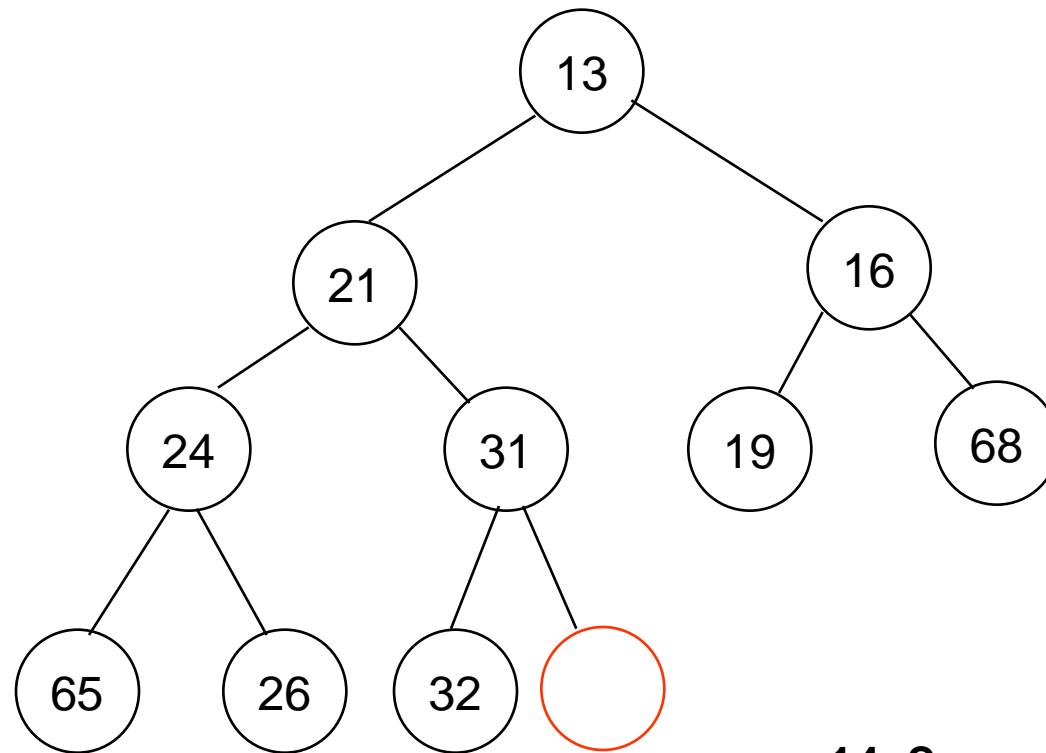
# Monceau – insertion de 14



0 1 2 3 4 5 6 7 8 9 10 11

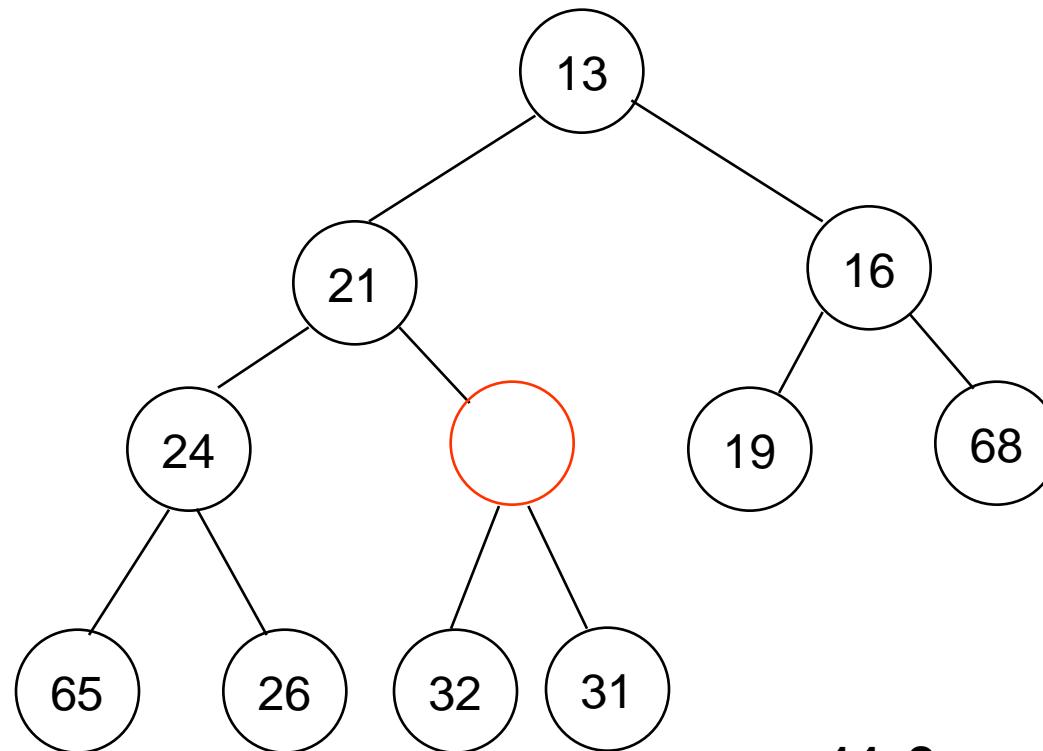
	13	21	16	24	31	19	68	65	26	32	
--	----	----	----	----	----	----	----	----	----	----	--

# Monceau - insertion de 14



	0	1	2	3	4	5	6	7	8	9	10	11
	13	21	16	24	31	19	68	65	26	32		

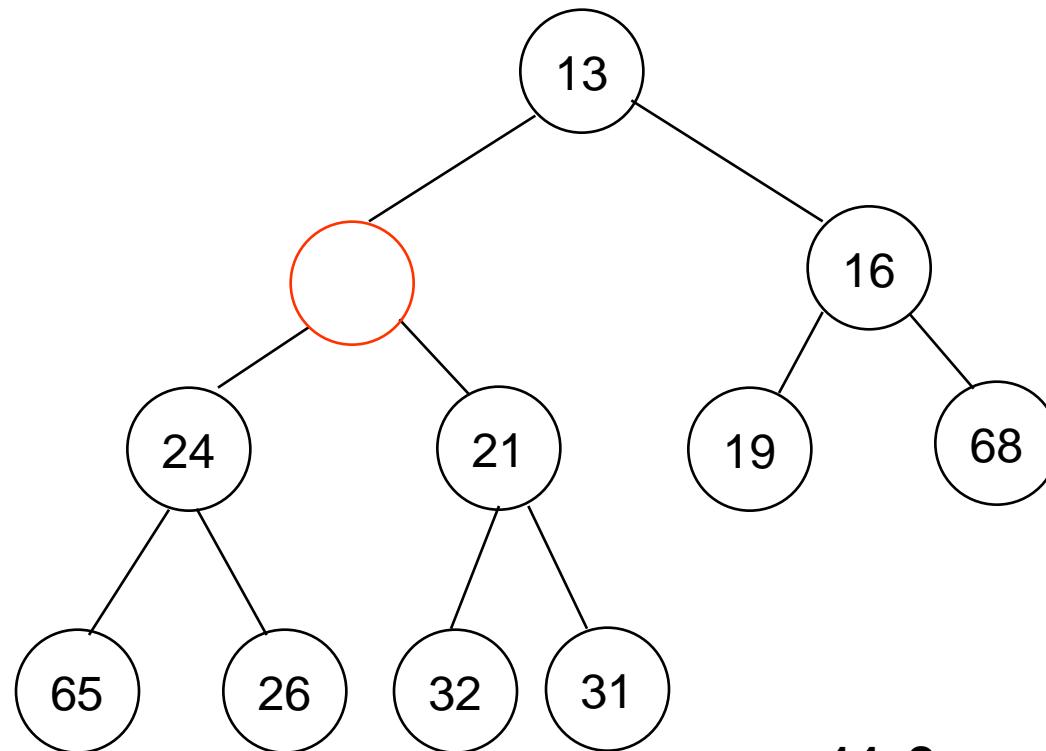
# Monceau - insertion de 14



0 1 2 3 4 5 6 7 8 9 10 11

	13	21	16	24		19	68	65	26	32	31
--	----	----	----	----	--	----	----	----	----	----	----

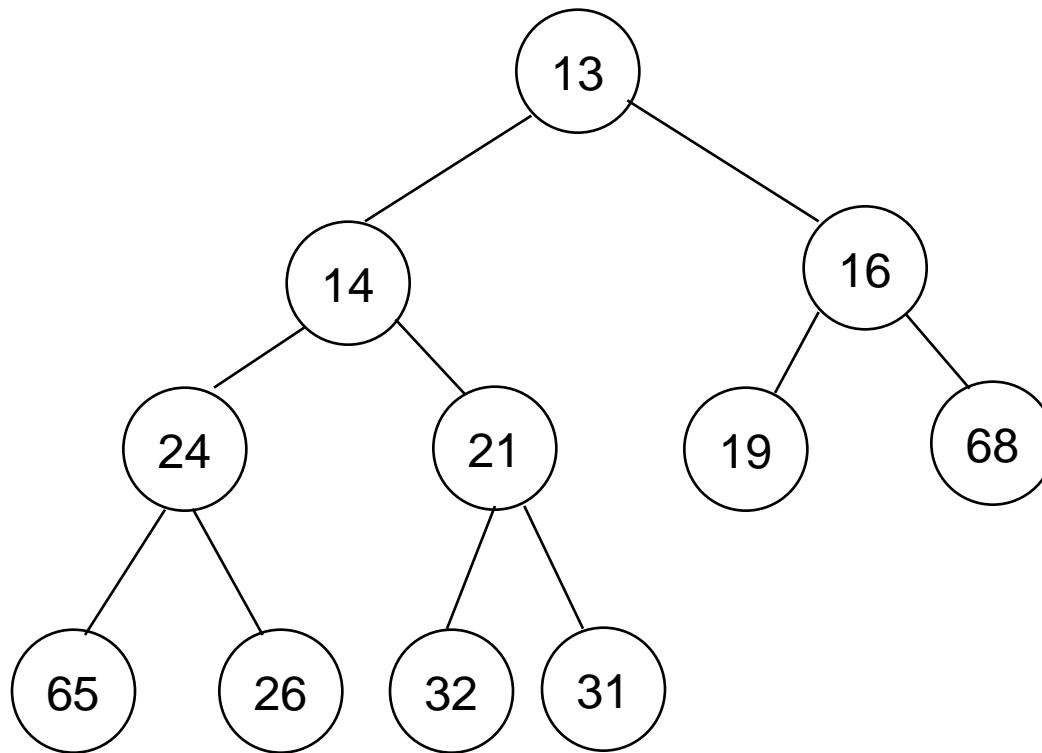
# Monceau - insertion de 14



0 1 2 3 4 5 6 7 8 9 10 11

	13		16	24	21	19	68	65	26	32	31
--	----	--	----	----	----	----	----	----	----	----	----

# Monceau - insertion de 14



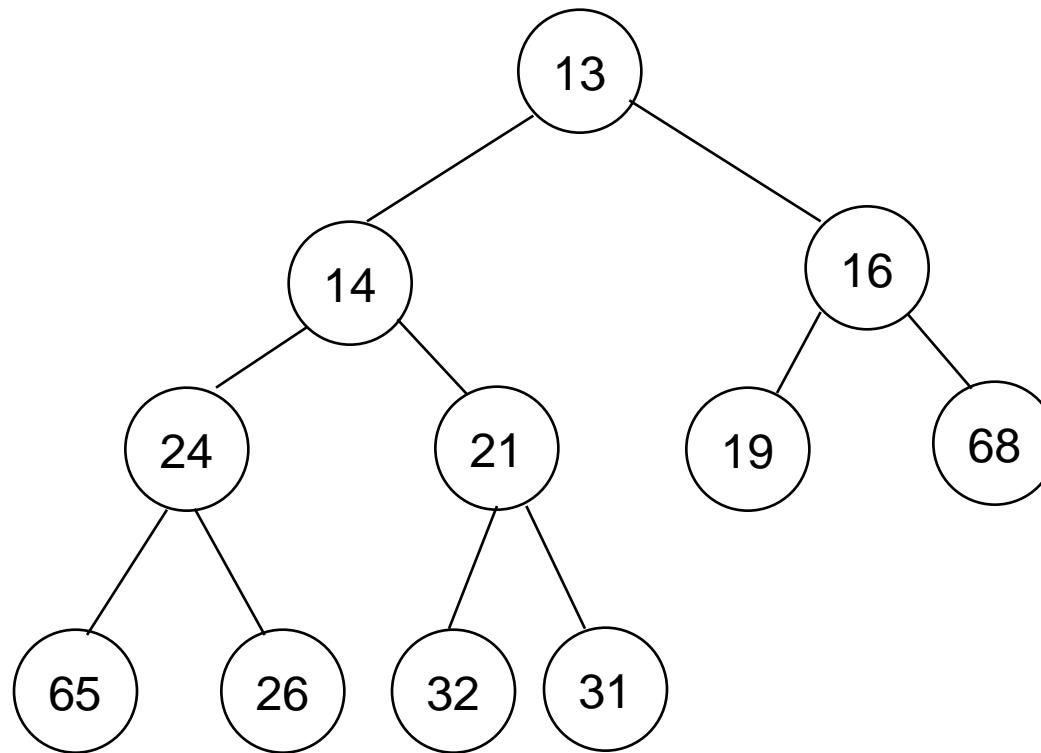
0 1 2 3 4 5 6 7 8 9 10 11

	13	14	16	24	21	19	68	65	26	32	31
--	----	----	----	----	----	----	----	----	----	----	----

# Insertion

```
/**  
 * Insert into the priority queue, maintaining heap order.  
 * Duplicates are allowed.  
 * @param x the item to insert.  
 */  
public void insert( AnyType x )  
{  
    if( currentSize == array.length - 1 )  
        enlargeArray( array.length * 2 + 1 );  
  
    // Percolate up  
    int hole = ++currentSize;  
    for( ; hole > 1 && x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2)  
        array[ hole ] = array[ hole / 2 ];  
    array[ hole ] = x;  
}
```

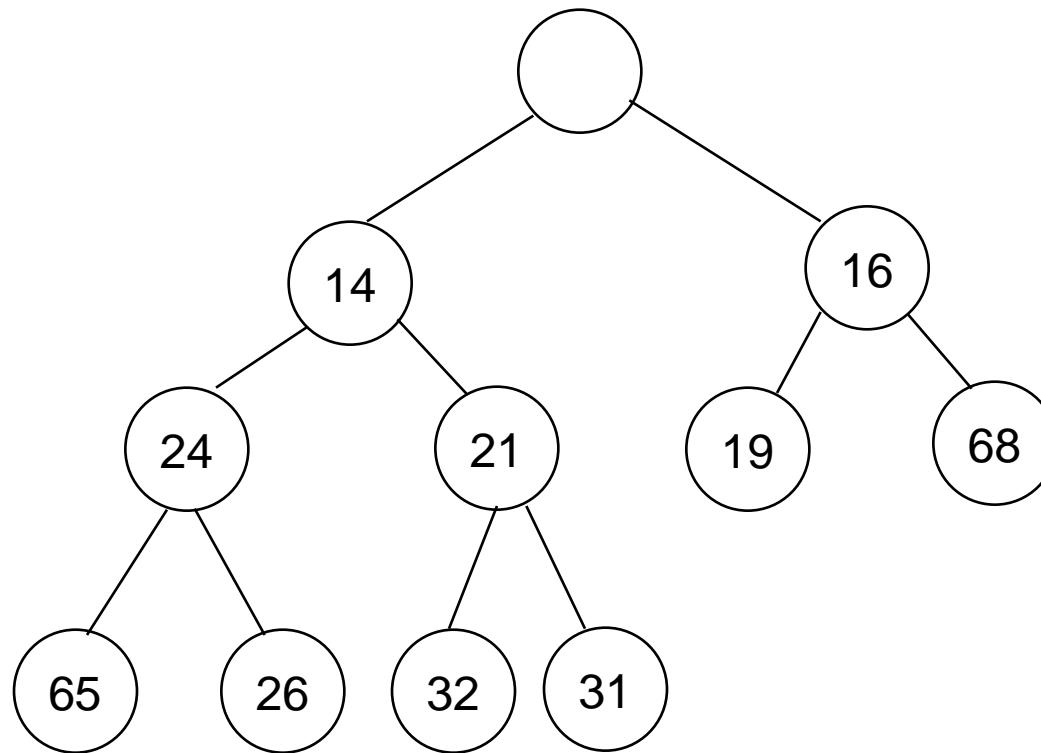
# Monceau - retrait



0    1    2    3    4    5    6    7    8    9    10    11

	13	14	16	24	21	19	68	65	26	32	31
--	----	----	----	----	----	----	----	----	----	----	----

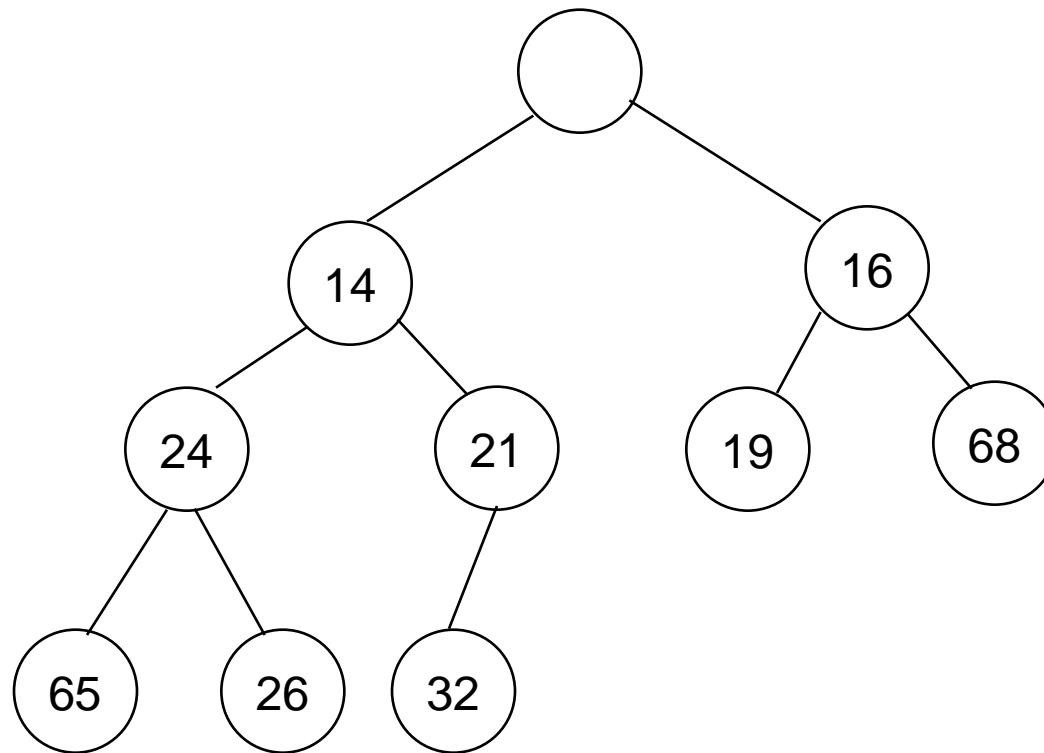
# Monceau - retrait



0 1 2 3 4 5 6 7 8 9 10 11

		14	16	24	21	19	68	65	26	32	31
--	--	----	----	----	----	----	----	----	----	----	----

# Monceau - retrait

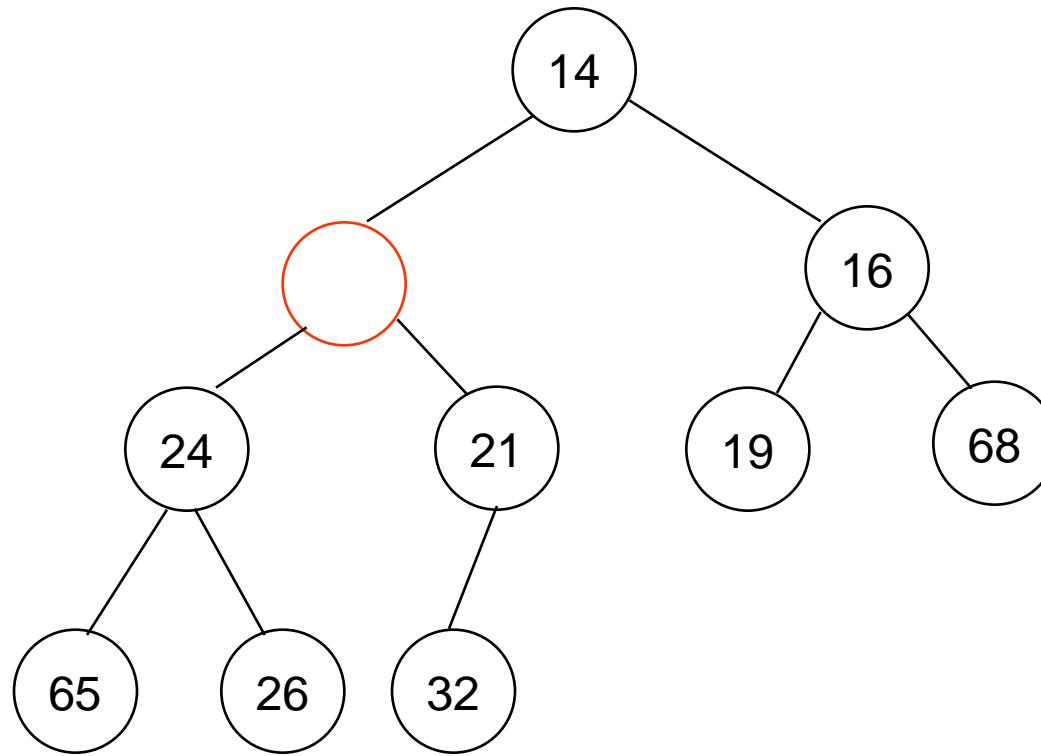


31 ?

0 1 2 3 4 5 6 7 8 9 10 11

		14	16	24	21	19	68	65	26	32	
--	--	----	----	----	----	----	----	----	----	----	--

# Monceau - retrait

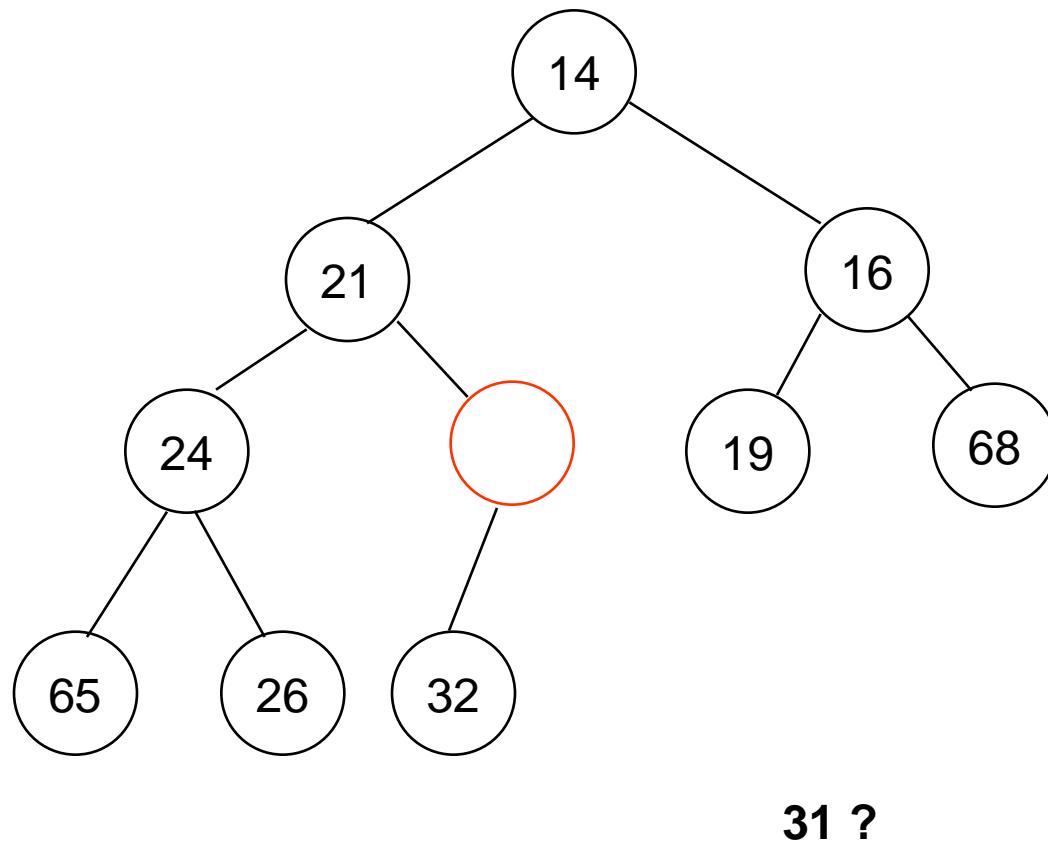


31 ?

0 1 2 3 4 5 6 7 8 9 10 11

	14		16	24	21	19	68	65	26	32	
--	----	--	----	----	----	----	----	----	----	----	--

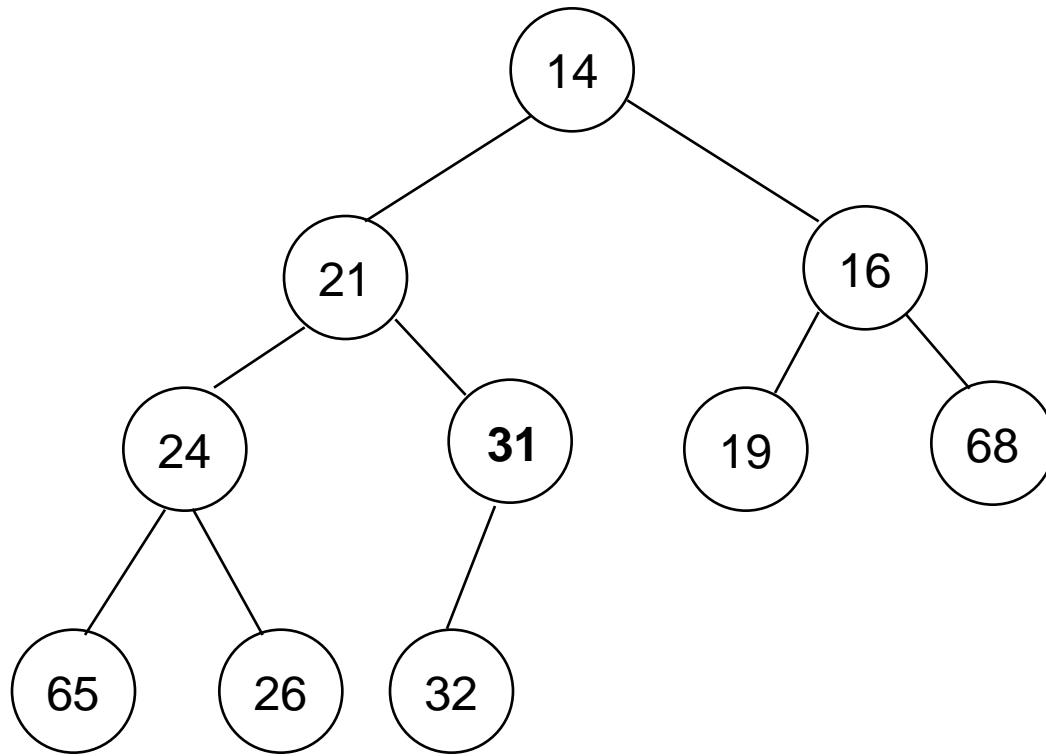
# Monceau - retrait



0 1 2 3 4 5 6 7 8 9 10 11

	14	21	16	24		19	68	65	26	32	
--	----	----	----	----	--	----	----	----	----	----	--

# Monceau - retrait



0 1 2 3 4 5 6 7 8 9 10 11

	14	21	16	24	31	19	68	65	26	32	
--	----	----	----	----	----	----	----	----	----	----	--

# Monceau - retrait

```
/**  
 * Remove the smallest item from the priority queue.  
 * @return the smallest item, or throw UnderflowException, if empty.  
 */  
public AnyType deleteMin( )  
{  
    if( isEmpty( ) )  
        throw new UnderflowException( );  
  
    AnyType minItem = findMin( );  
    array[ 1 ] = array[ currentSize-- ];  
    percolateDown( 1 );  
  
    return minItem;  
}
```

# Monceau - retrait

```
/**  
 * Internal method to percolate down in the heap.  
 * @param hole the index at which the percolate begins.  
 */  
private void percolateDown( int hole )  
{  
    int child;  
    AnyType tmp = array[ hole ];  
  
    for( ; hole * 2 <= currentSize; hole = child )  
    {  
        child = hole * 2; //Considérer fils de gauche  
  
        if( child != currentSize && // il y a deux fils  
            array[ child + 1 ].compareTo( array[ child ] ) < 0 ) //et fils droit<fils gauche  
            child++; //Considérer fils droit  
        if( array[ child ].compareTo( tmp ) < 0 )//fils considéré< élément à percoler  
            array[ hole ] = array[ child ];//Remonter le fils courrent de un niveau  
        else  
            break; //sortir de la boucle. L'élément à percoler sera inséré à position hole  
    }  
  
    array[ hole ] = tmp; // Insérer l'élément à percoler à la position hole  
}
```

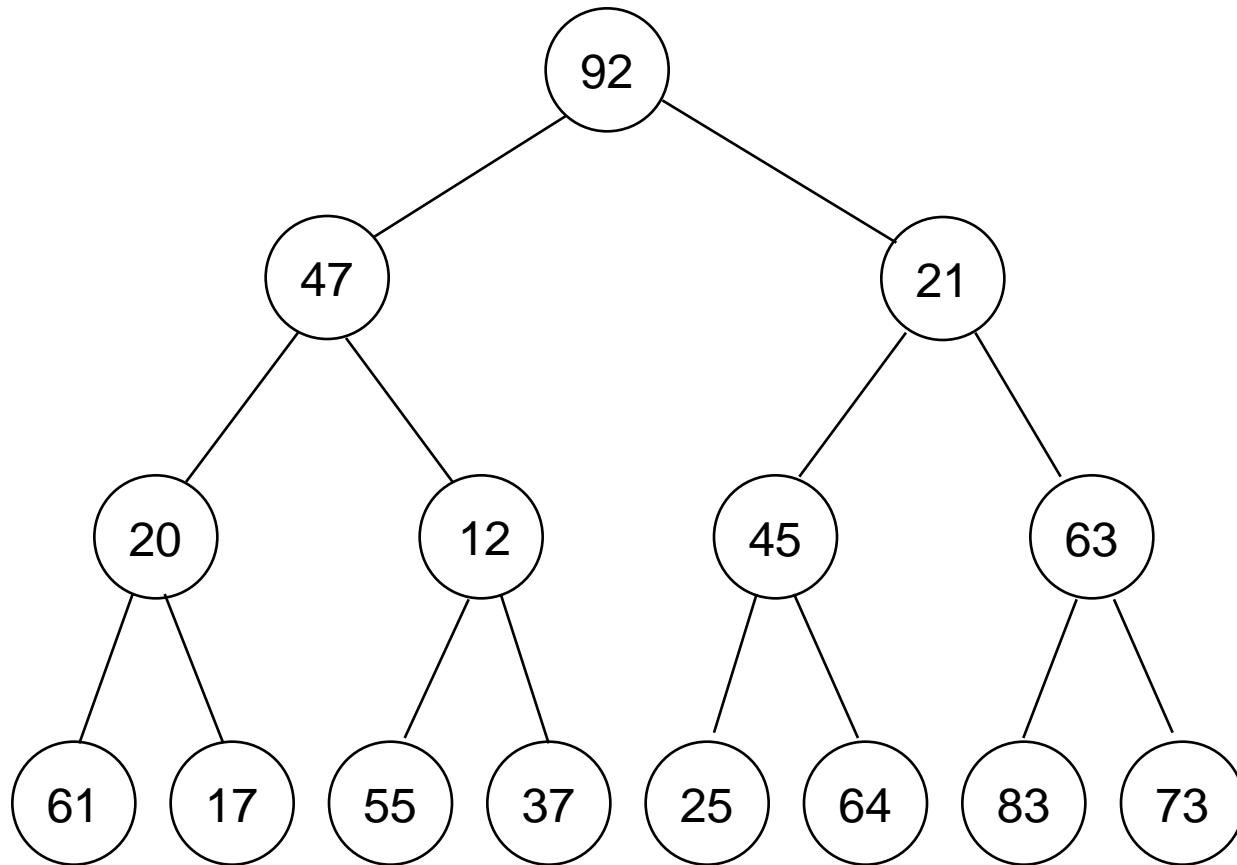
# Monceau – insertion et retrait

- La complexité en pire cas d'un retrait et d'une insertion est  $O(\log(n))$
- *La complexité* en meilleur cas d'un retrait et d'une insertion est  $O(1)$
- La complexité moyenne d'un retrait :  $O(\log(n))$
- La complexité moyenne d'une insertion est :  $O(1)$

# Plan

- Définition de monceau et implémentation
- Insertion et retrait d'éléments
- **Construction d'un monceux**
- Tri par monceau

# Monceau - construction

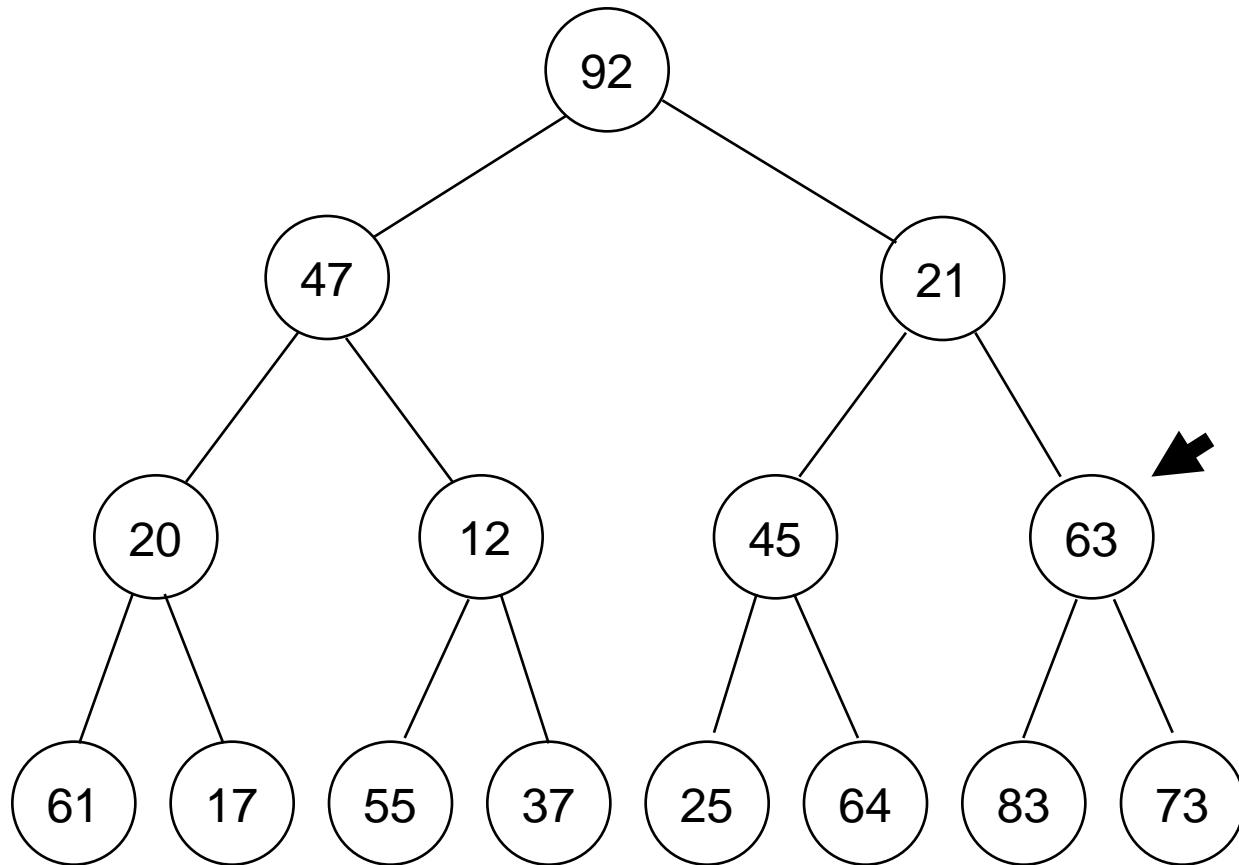


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	20	12	45	63	61	17	55	37	25	64	83	73

# Construction

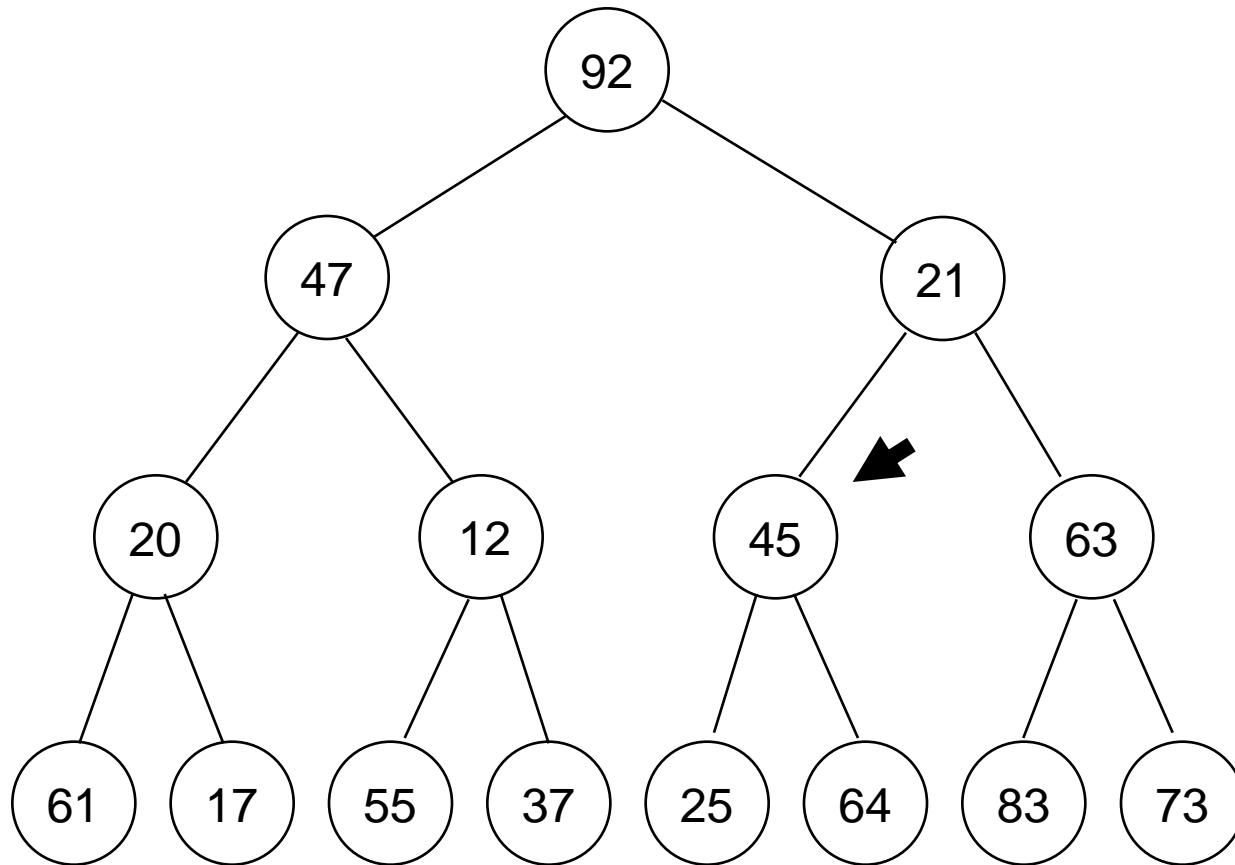
```
/**  
 * Establish heap order property from an arbitrary  
 * arrangement of items. Runs in linear time.  
 */  
private void buildHeap( )  
{  
    for( int i = currentSize / 2; i > 0; i-- )  
        percolateDown( i );  
}
```

# Monceau - construction



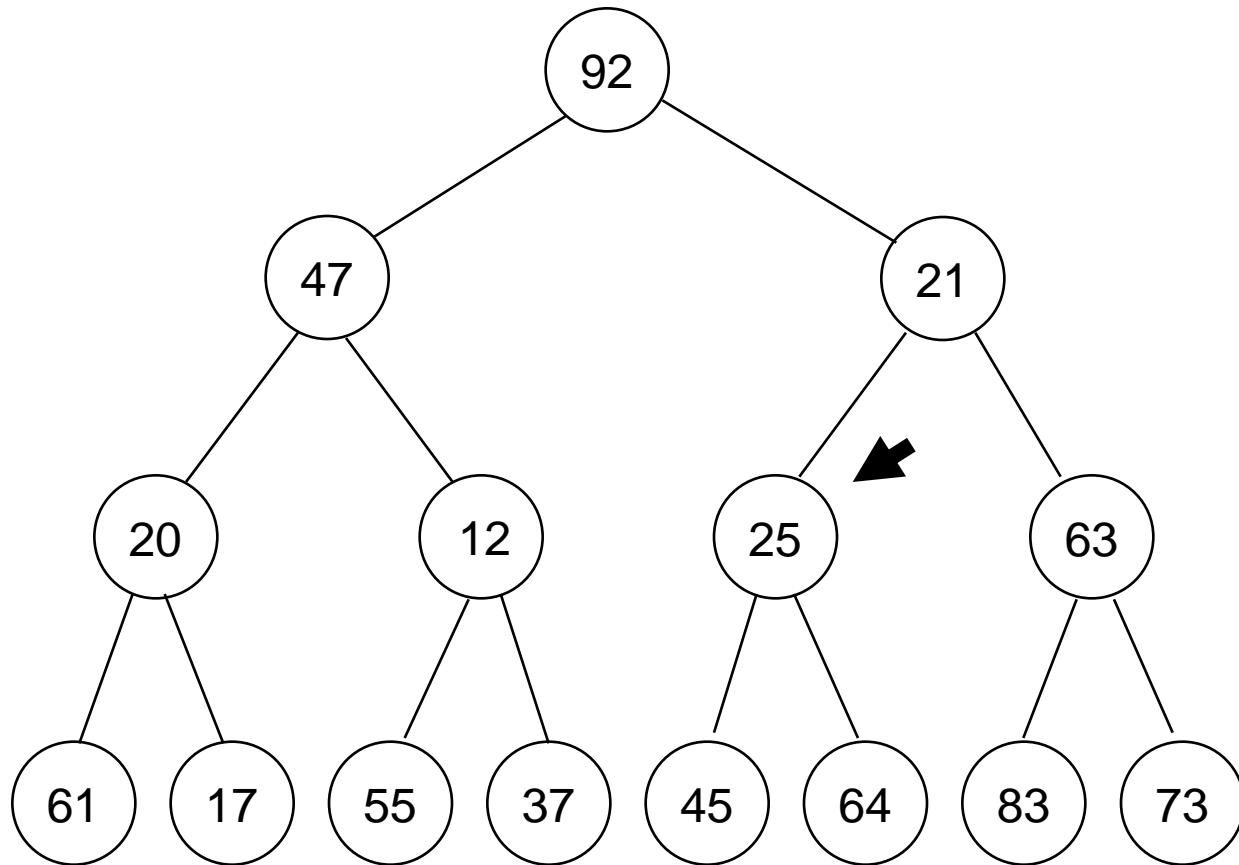
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	20	12	45	63	61	17	55	37	25	64	83	73

# Monceau - construction



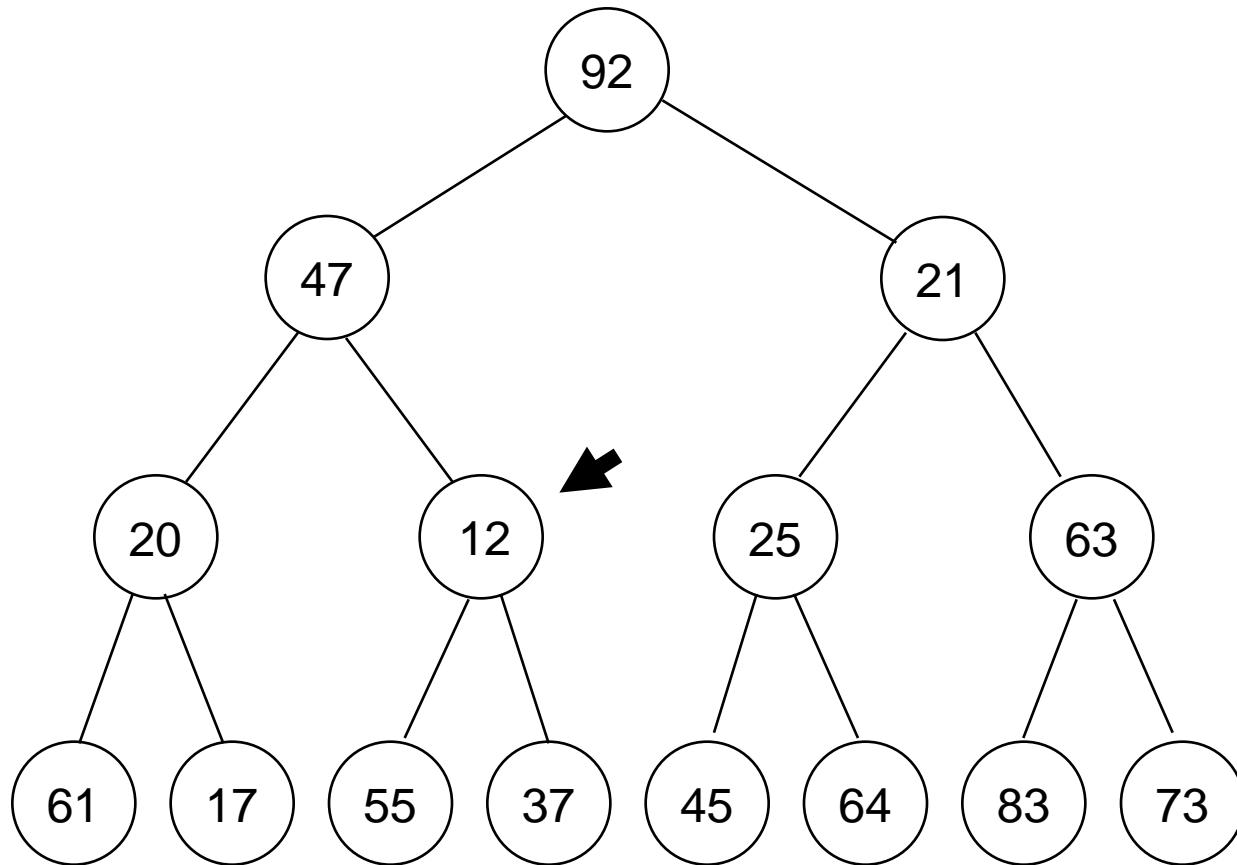
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	20	12	45	63	61	17	55	37	25	64	83	73

# Monceau - construction



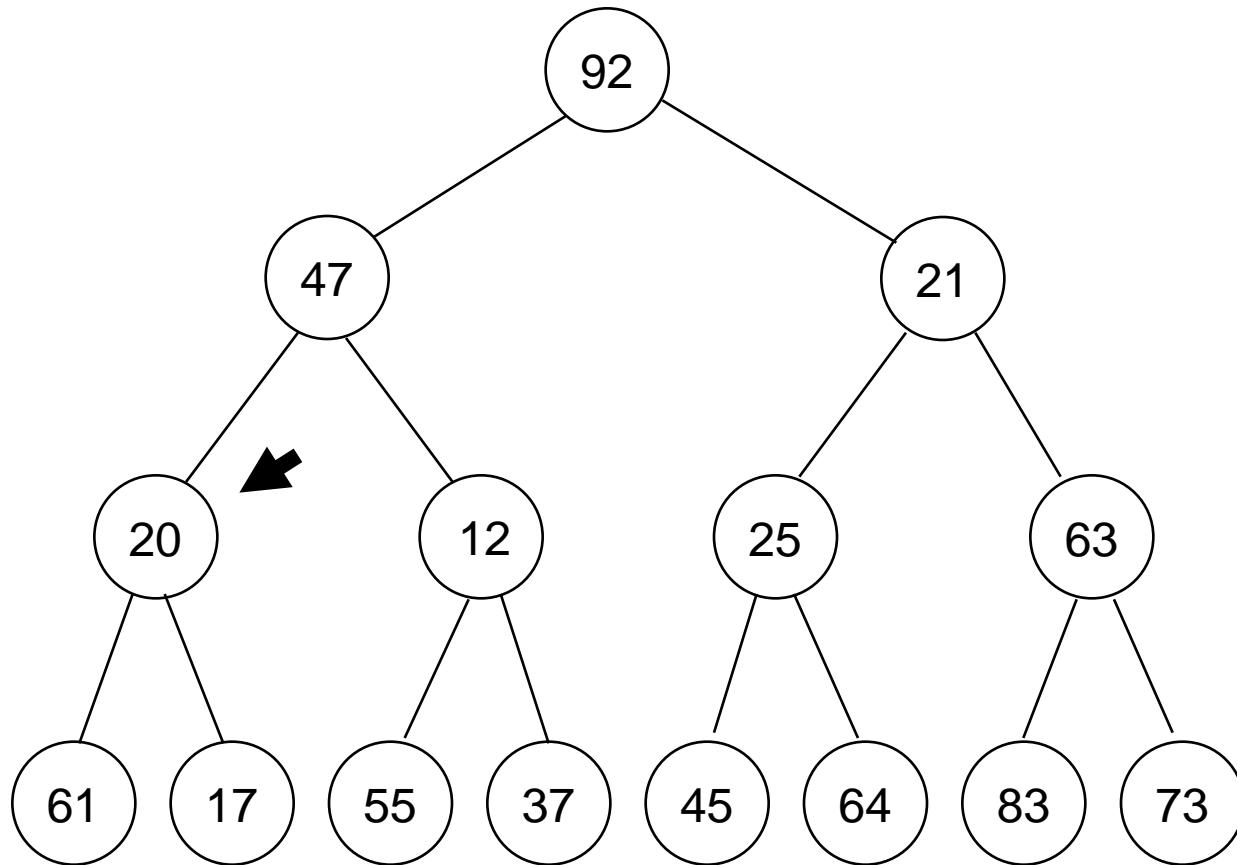
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	20	12	25	63	61	17	55	37	45	64	83	73

# Monceau - construction



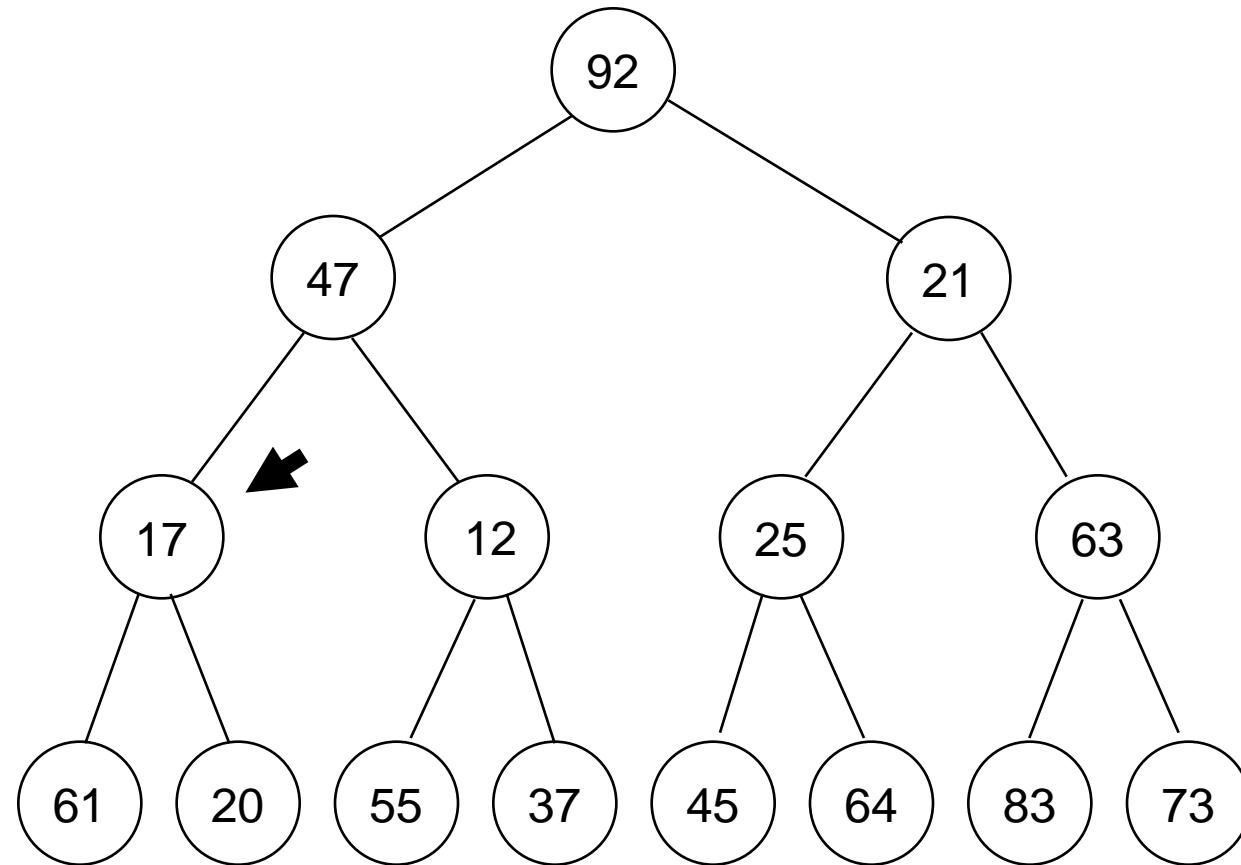
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	20	12	25	63	61	17	55	37	45	64	83	73

# Monceau - construction



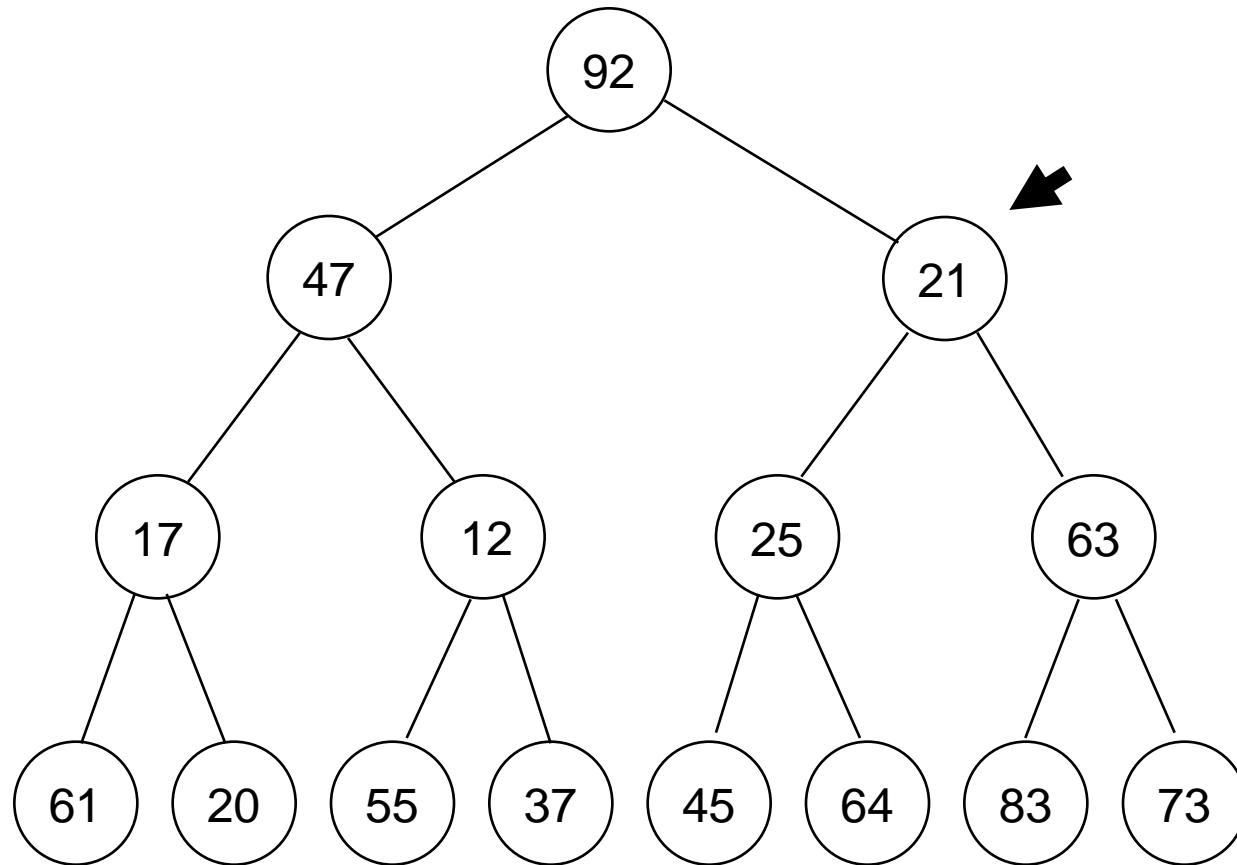
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	20	12	25	63	61	17	55	37	45	64	83	73

# Monceau - construction



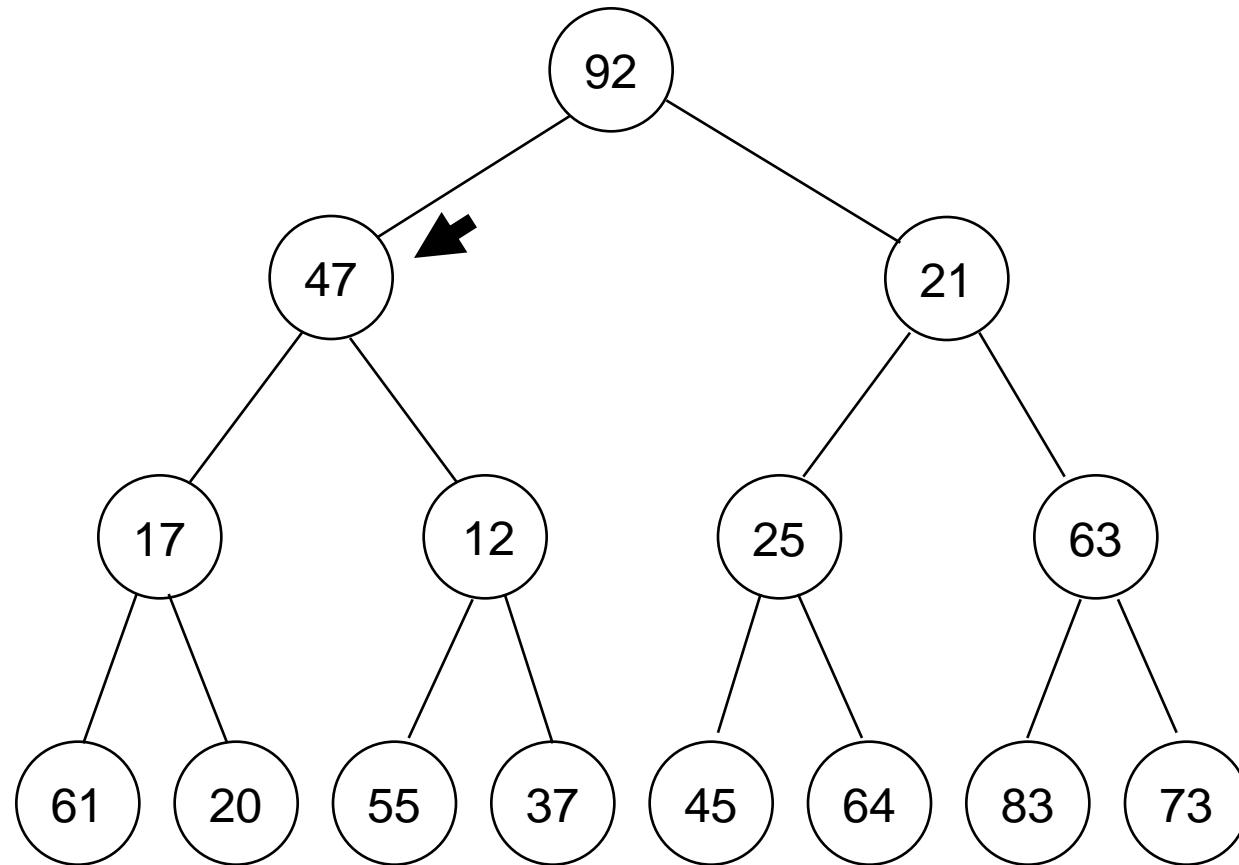
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	17	12	25	63	61	20	55	37	45	64	83	73

# Monceau - construction



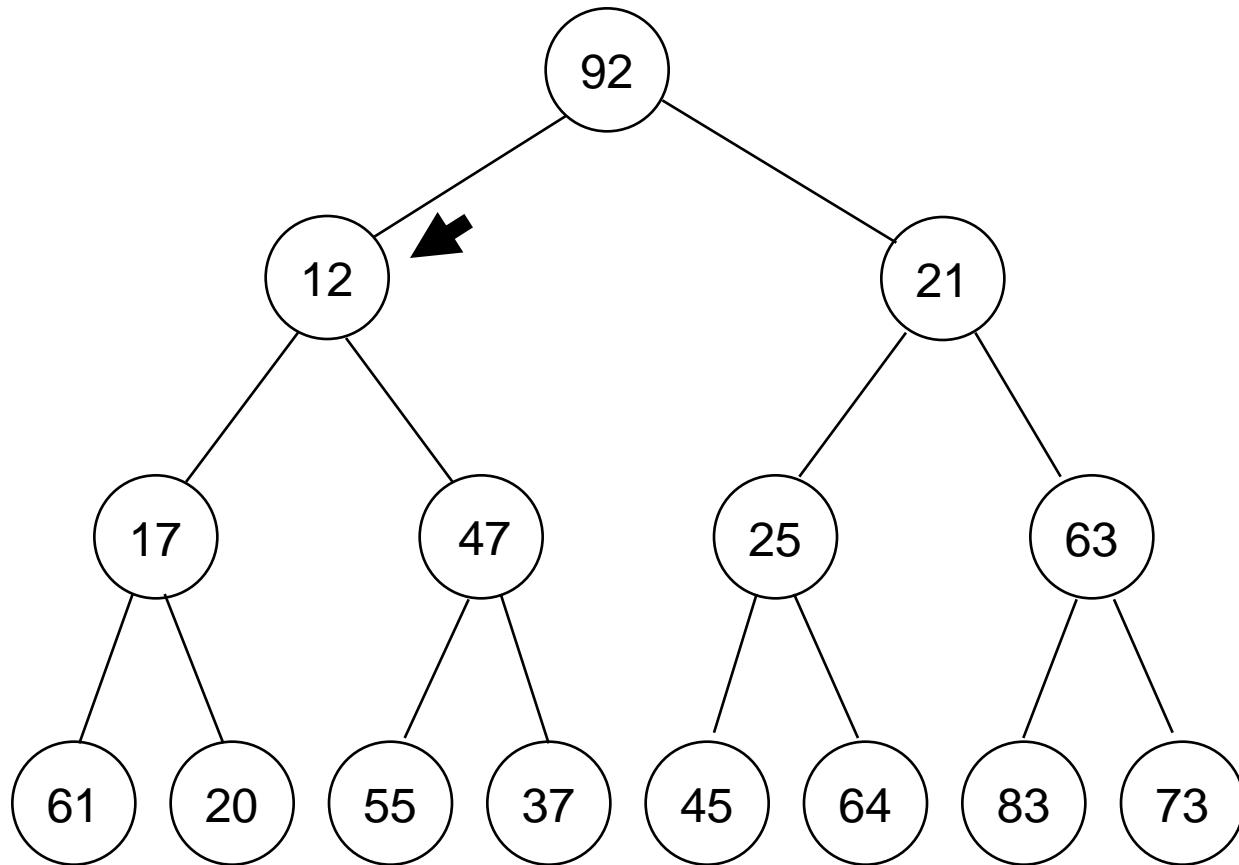
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	17	12	25	63	61	20	55	37	45	64	83	73

# Monceau - construction



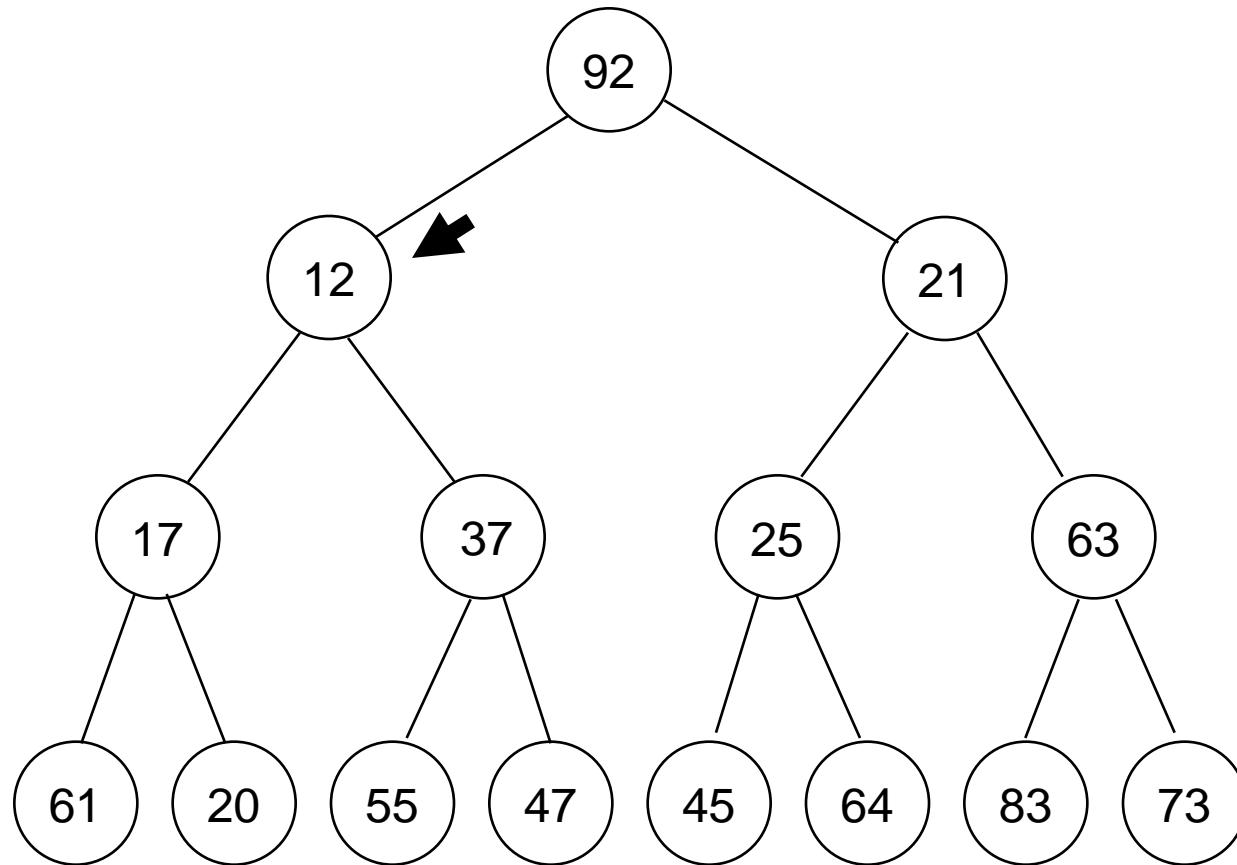
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	17	12	25	63	61	20	55	37	45	64	83	73

# Monceau - construction



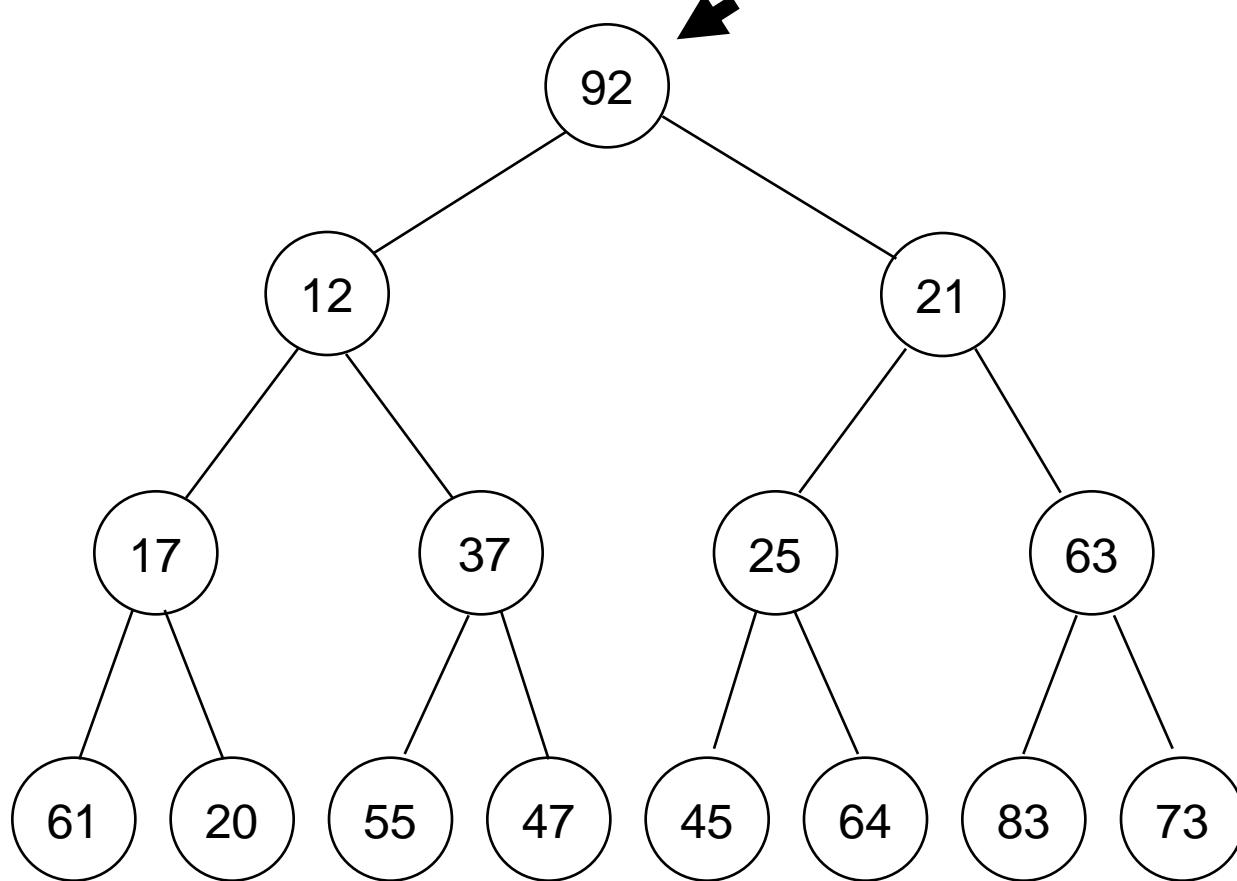
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	12	21	17	47	25	63	61	20	55	37	45	64	83	73

# Monceau - construction



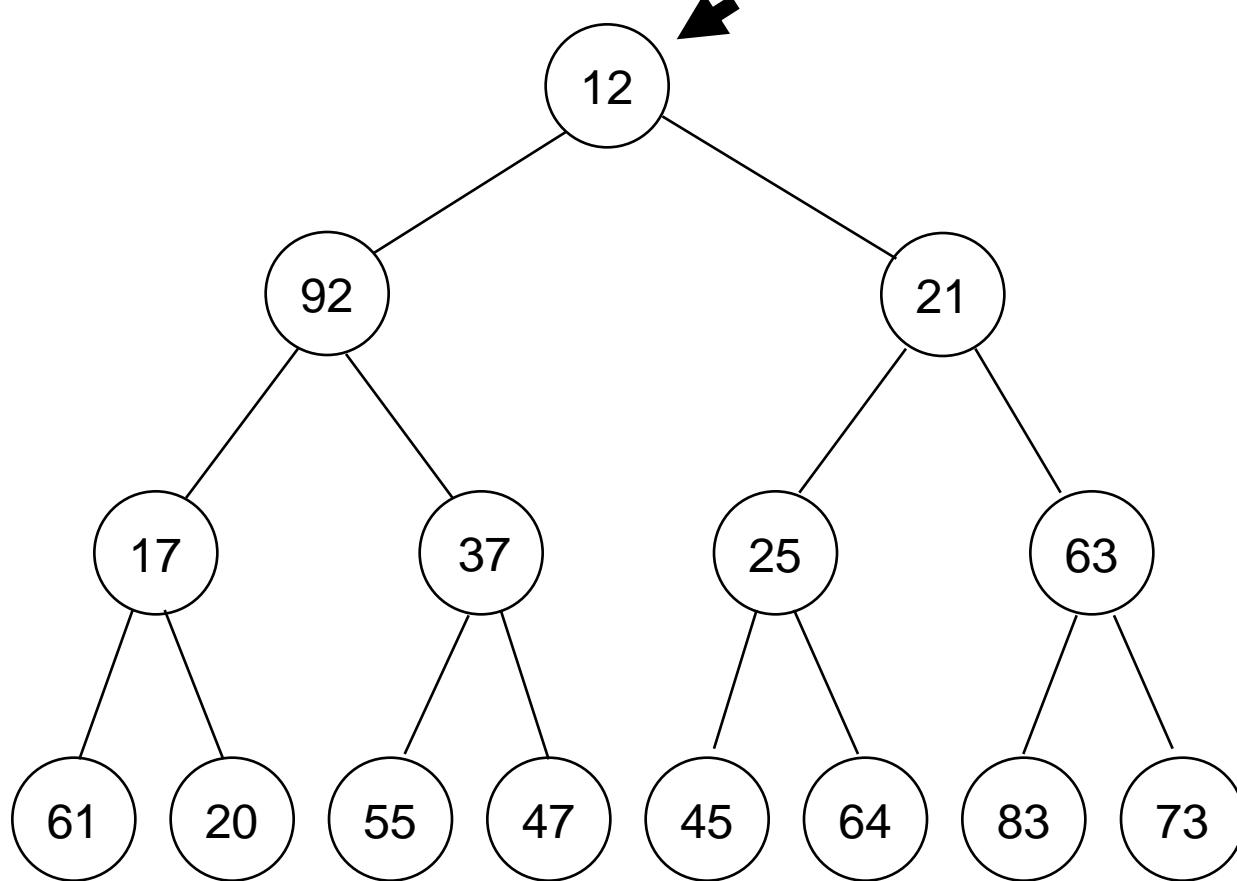
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	12	21	17	37	25	63	61	20	55	47	45	64	83	73

# Monceau - construction



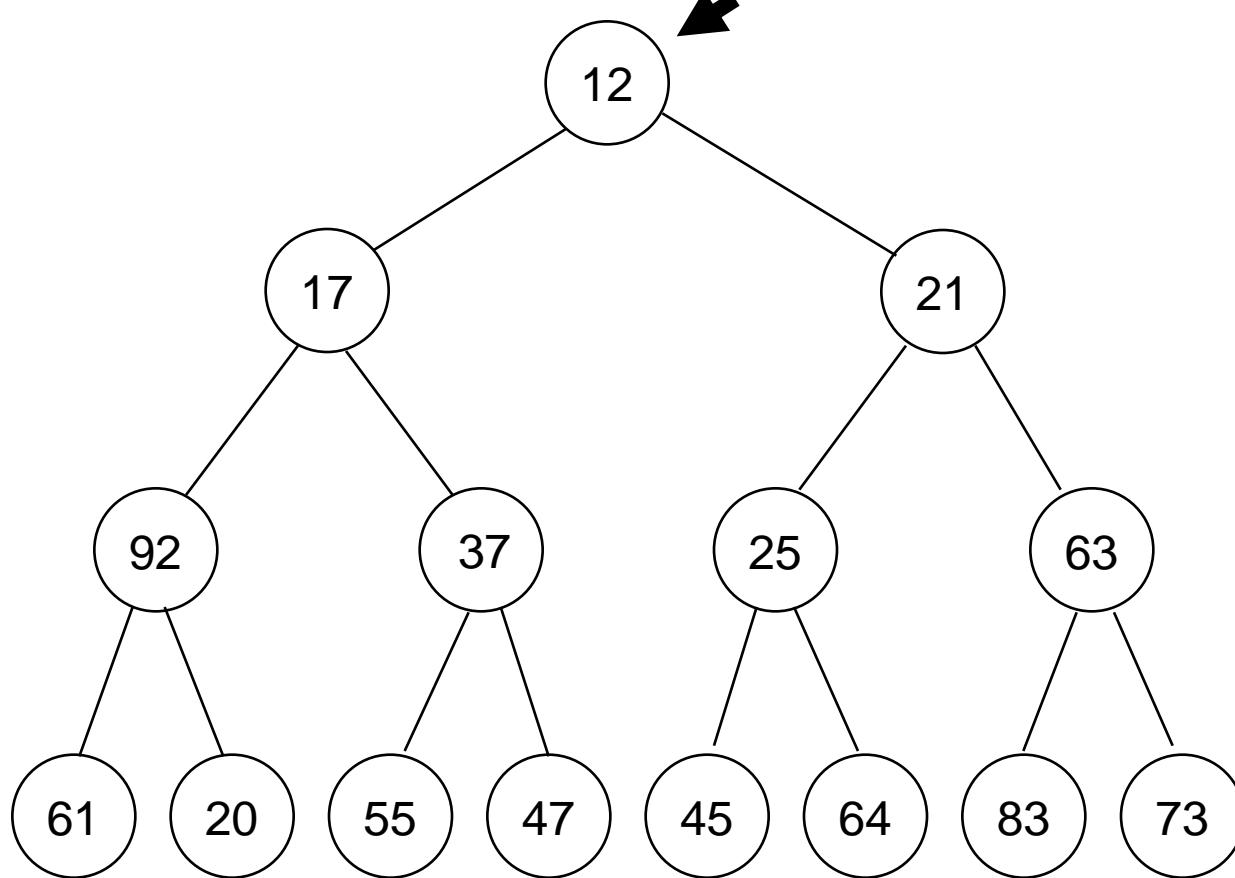
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	12	21	17	37	25	63	61	20	55	47	45	64	83	73

# Monceau - construction



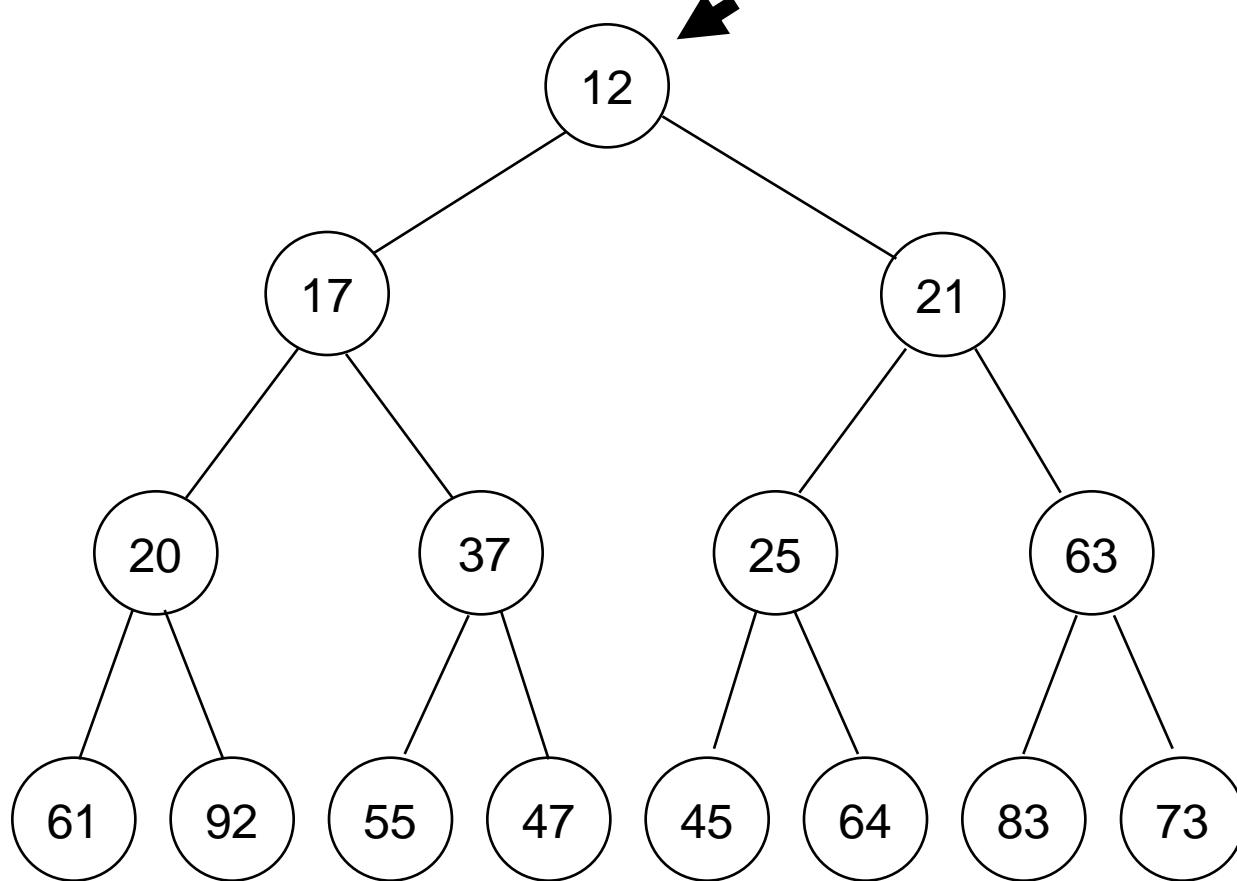
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	12	92	21	17	37	25	63	61	20	55	47	45	64	83	73

# Monceau - construction



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	12	17	21	92	37	25	63	61	20	55	47	45	64	83	73

# Monceau - construction



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	12	17	21	20	37	25	63	61	92	55	47	45	64	83	73

# Plan

- Définition de monceau et implémentation
- Insertion et retrait d'éléments
- Construction d'un monceux
- Tri par monceau

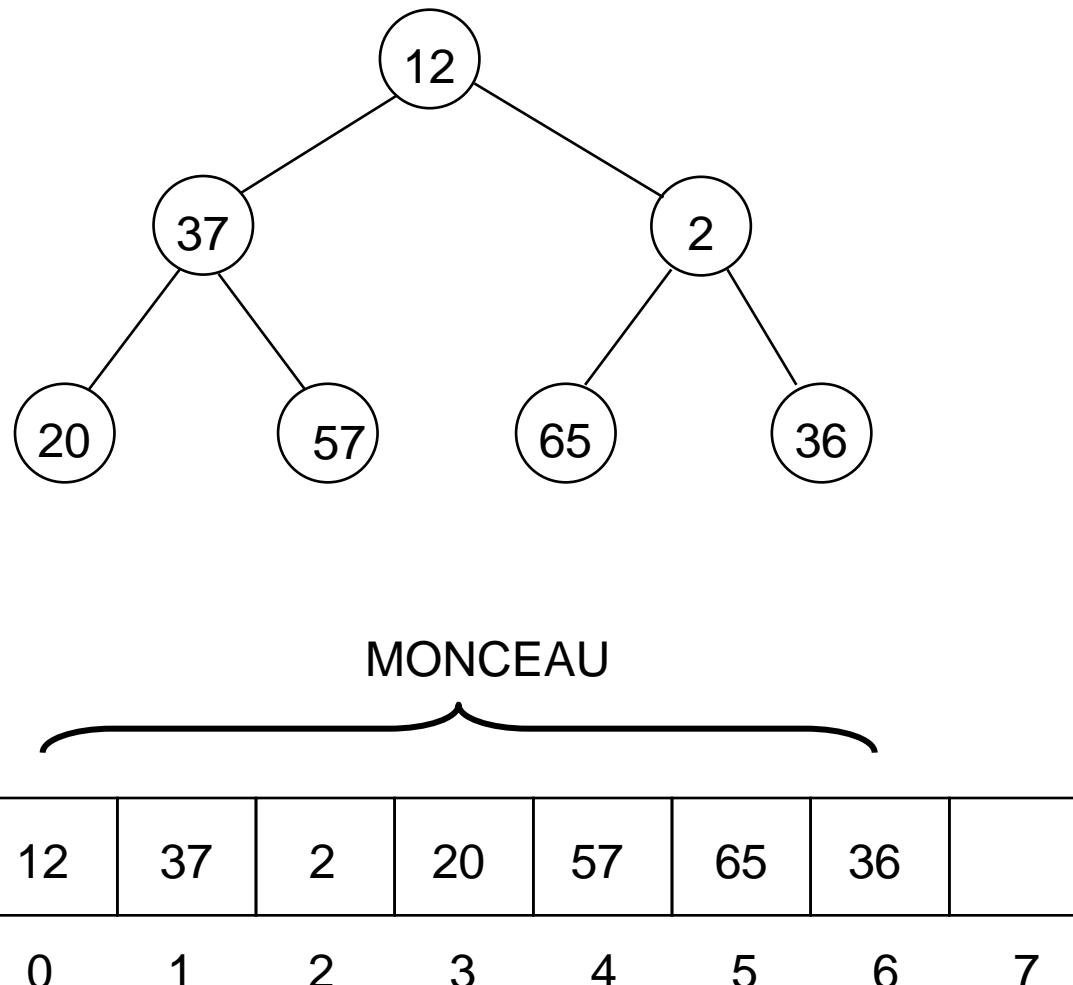
# Tri par monceau - Principe

- On applique les modifications suivantes au monceau:
  - chaque nœud a une valeur plus grande ou égale à celle de tous ses descendants
  - dans le tableau qui implémente le monceau, la racine se trouve à l'index 0 (au lieu de 1)
  - pour chaque nœud  $i$ , les index sont donc  $2*i + 1$  pour le fils gauche et  $2*i+2$  pour le fils droit

# Tri par monceau - Principe

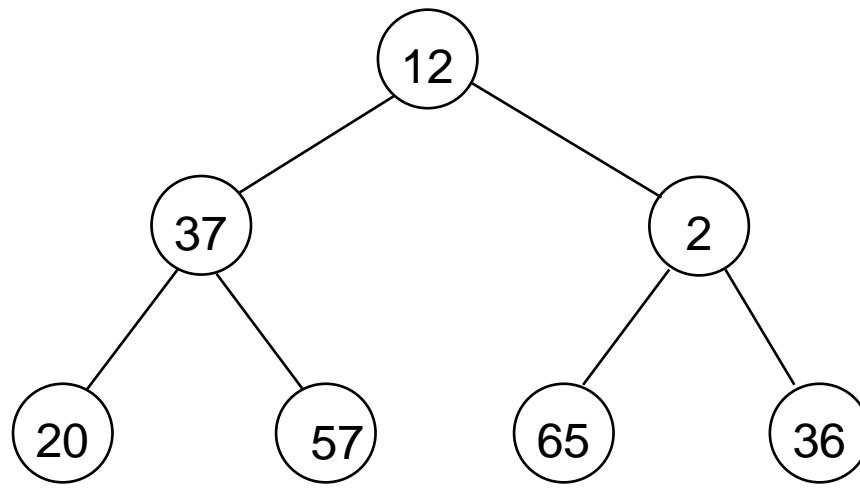
- La procédure est la suivante:
  1. On met dans le tableau les valeurs à trier
  2. On crée le monceau initial
  3. On retire la racine du monceau (qui est la plus grande valeur) et on la permute avec le dernier item du monceau
  4. On fait percoler la racine, s'il y a lieu
  5. On répète les étapes 3 et 4 avec le monceau obtenu qui contient maintenant un élément de moins

# Tri par monceau - Exemple

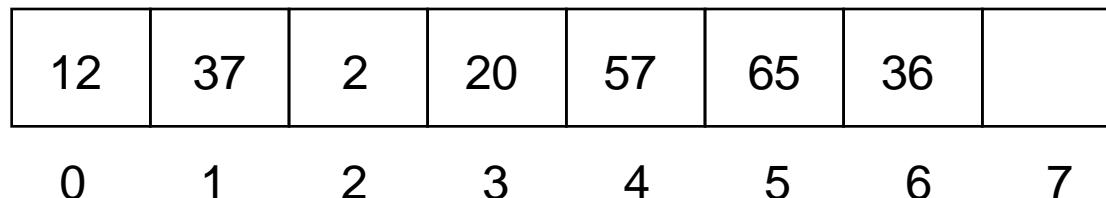


# Tri par monceau - Exemple

*Création du monceau*

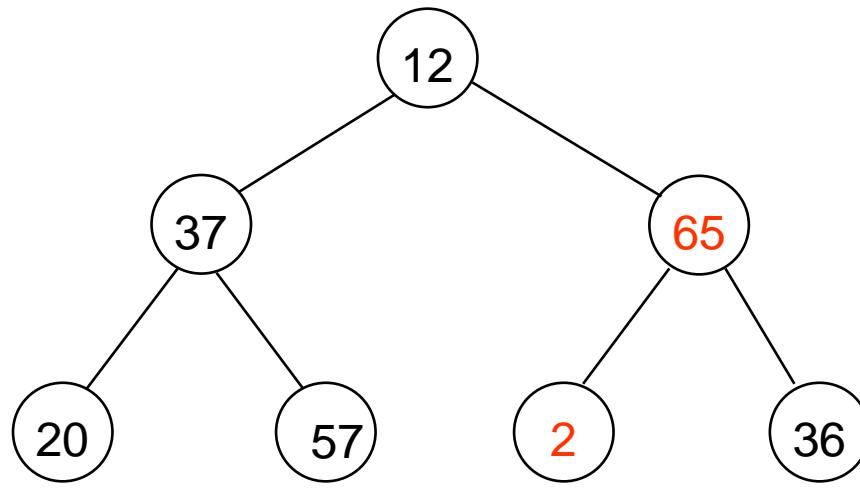


MONCEAU

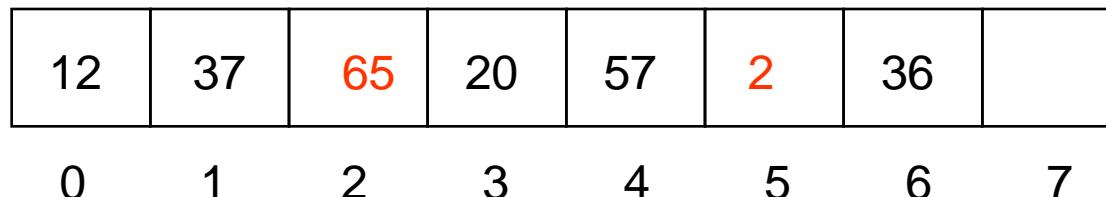


# Tri par monceau - Exemple

*Création du monceau*

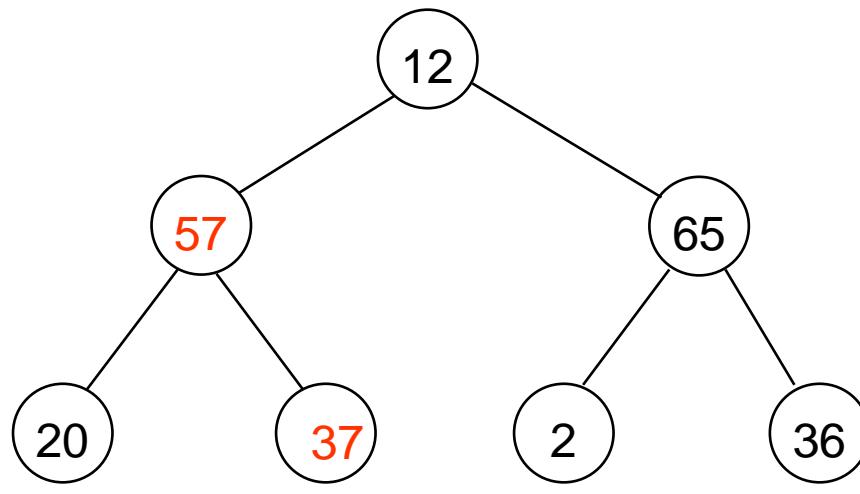


MONCEAU

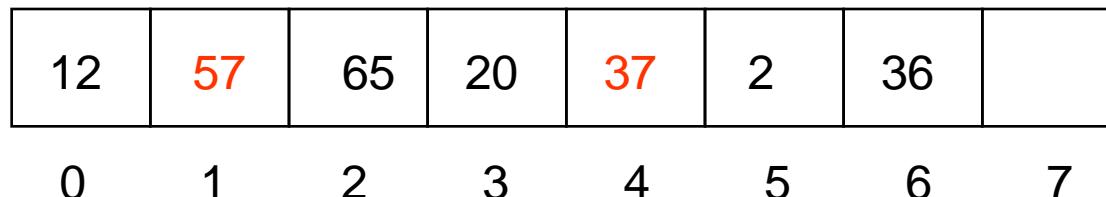


# Tri par monceau - Exemple

*Création du monceau*

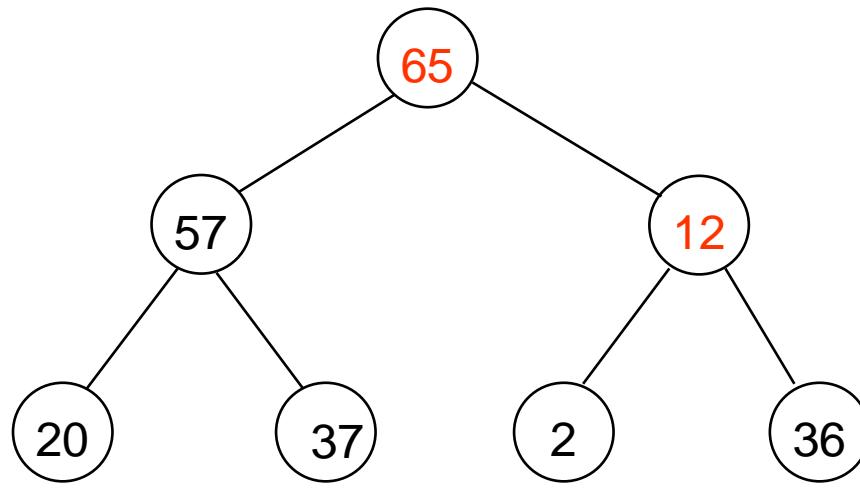


MONCEAU

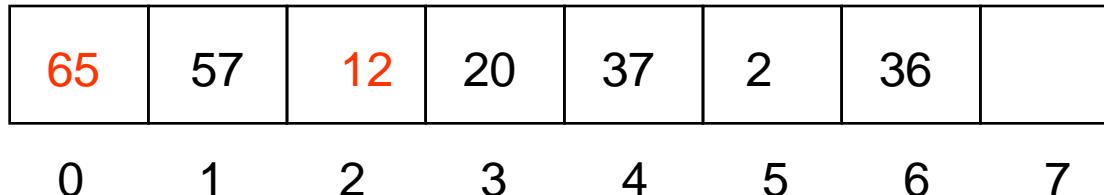


# Tri par monceau - Exemple

*Création du monceau*

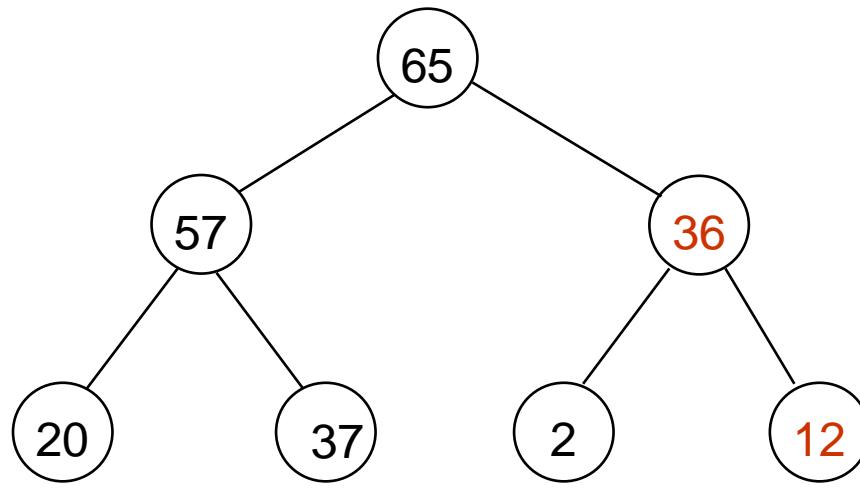


MONCEAU

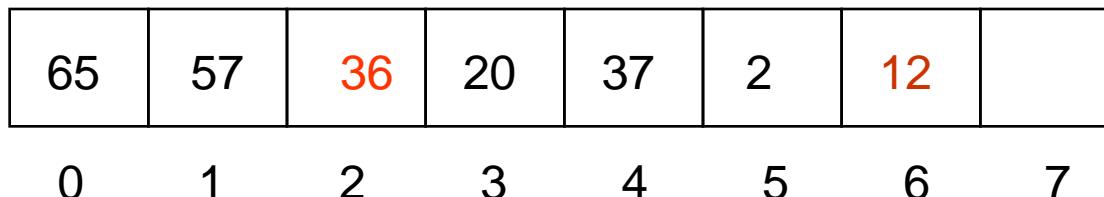


# Tri par monceau - Exemple

*Création du monceau*

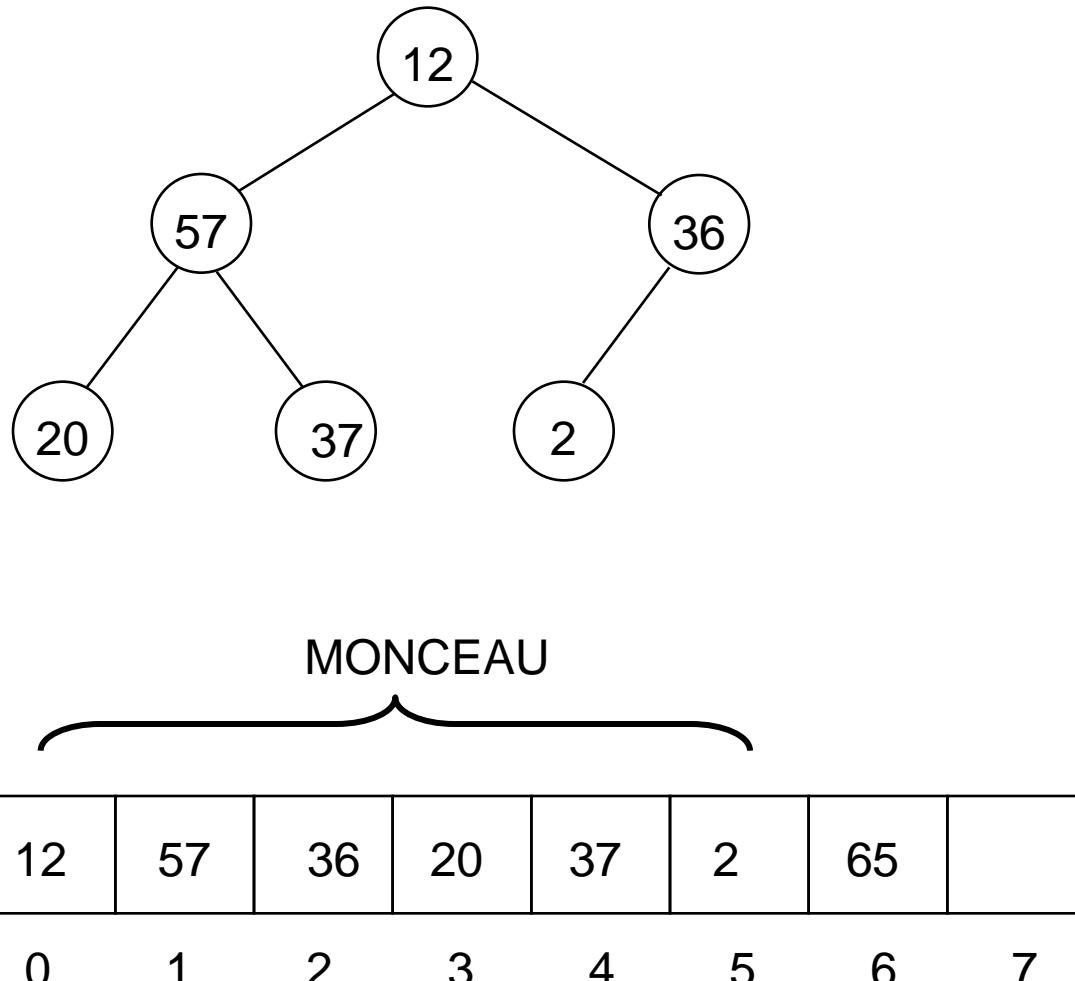


MONCEAU



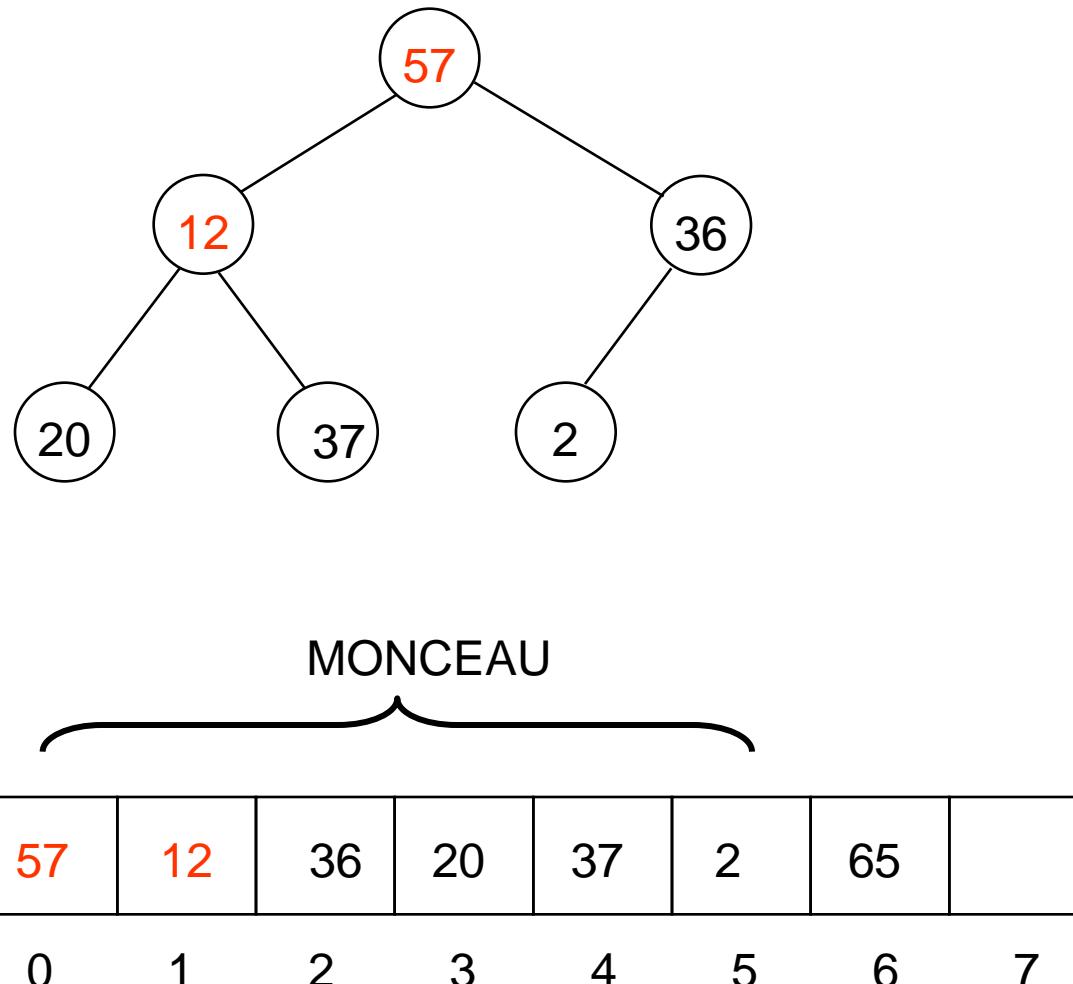
# Tri par monceau - Exemple

*Retrait du 1<sup>er</sup> élément*



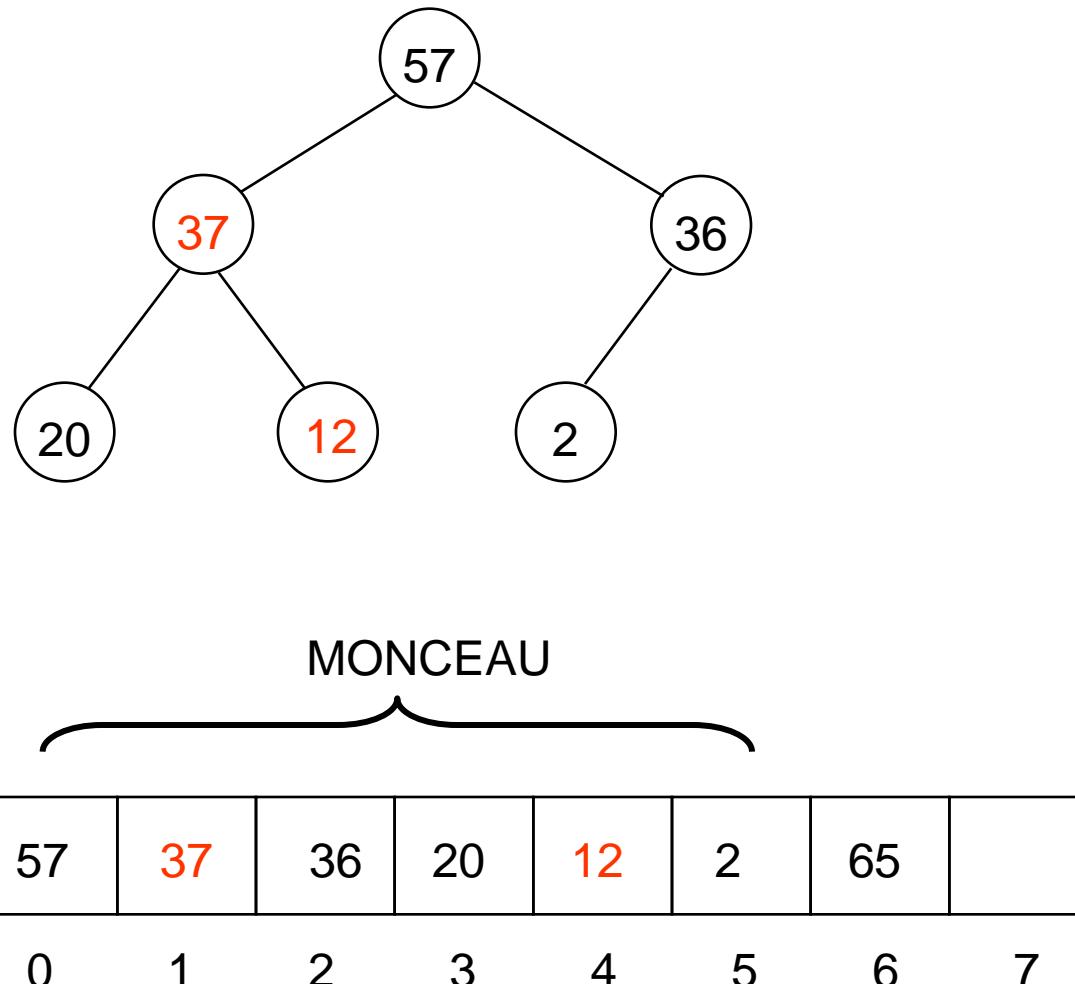
# Tri par monceau - Exemple

Percoler



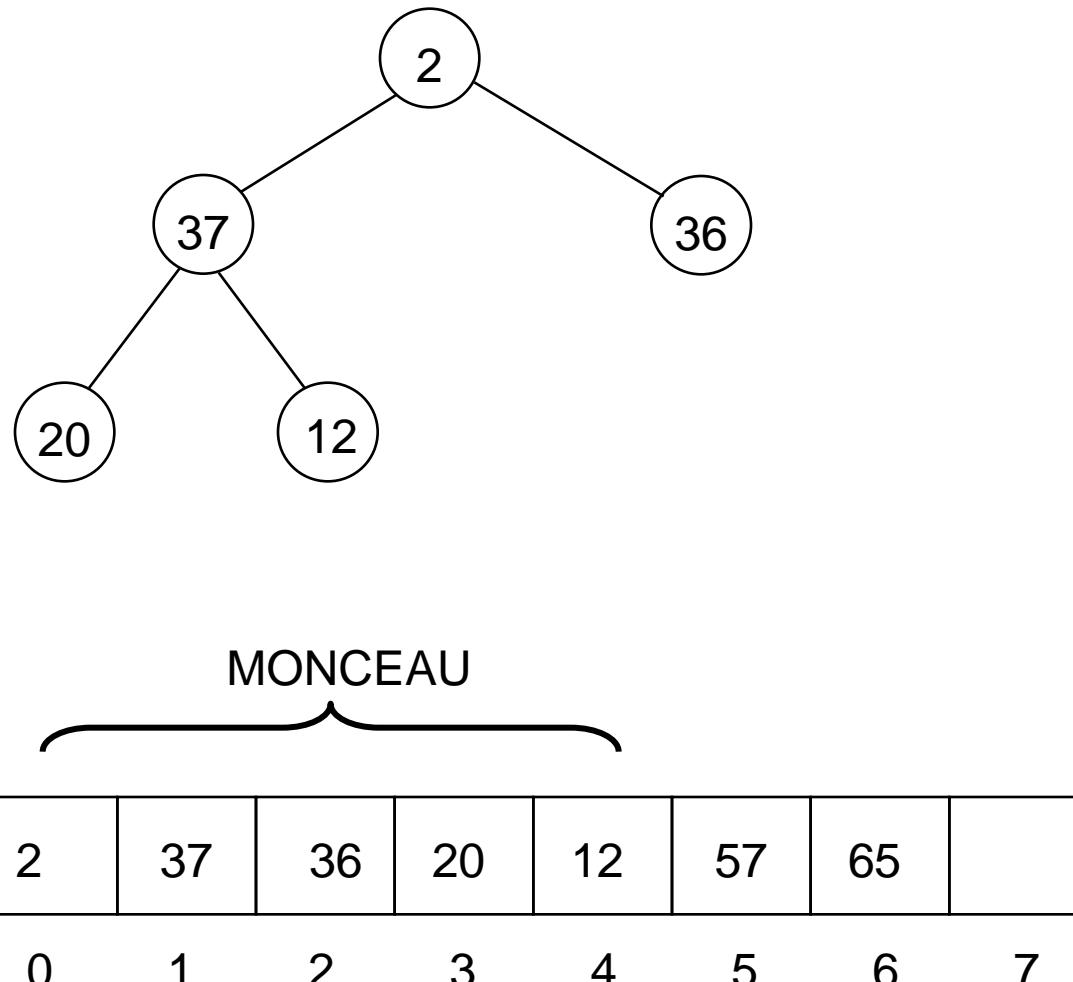
# Tri par monceau - Exemple

*Percoler*



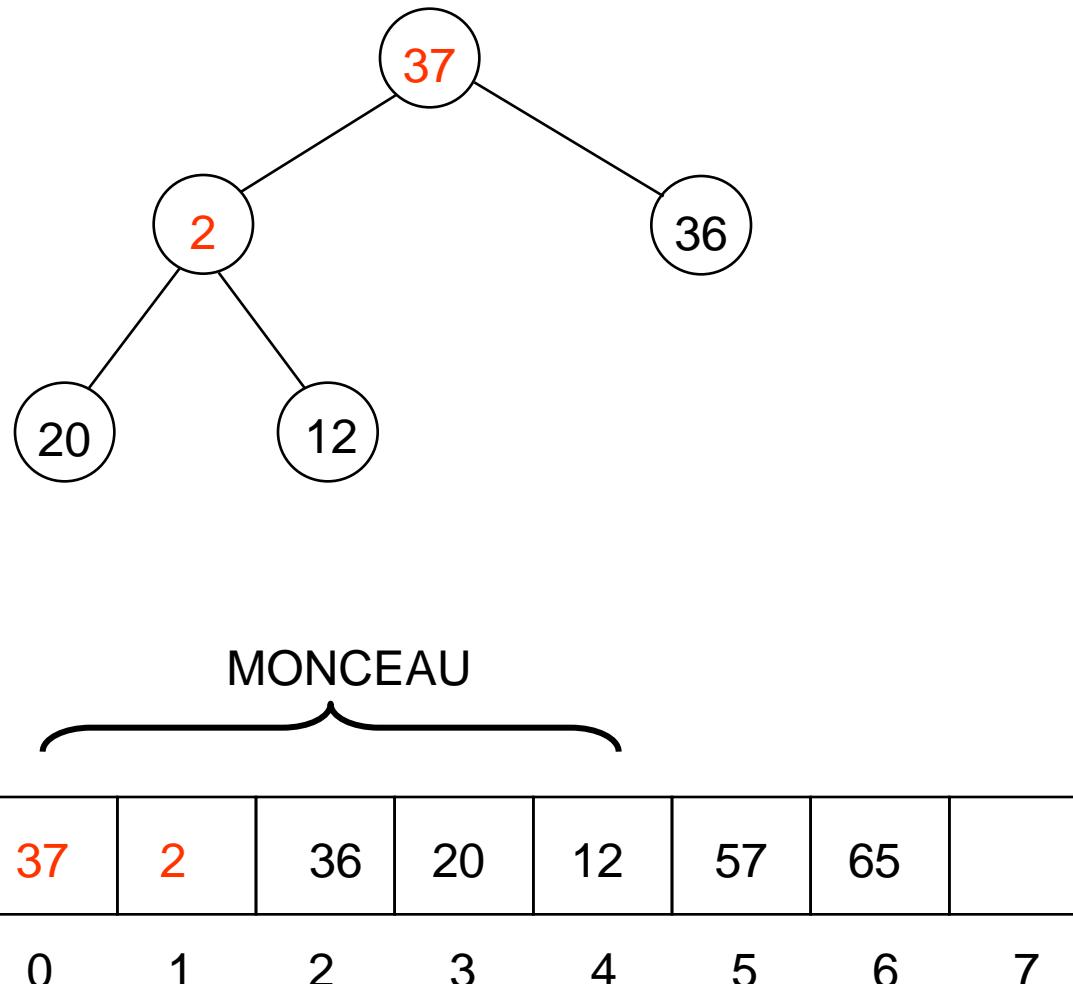
# Tri par monceau - Exemple

*Retrait du 2<sup>ème</sup> élément*



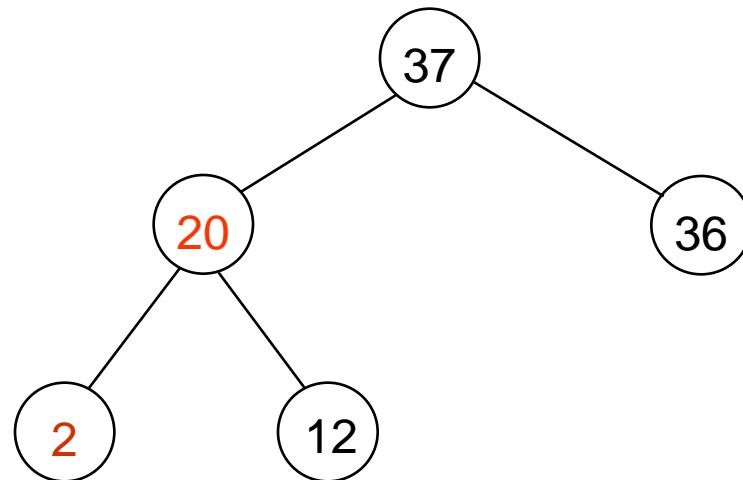
# Tri par monceau - Exemple

Percoler



# Tri par monceau - Exemple

*Percoler*

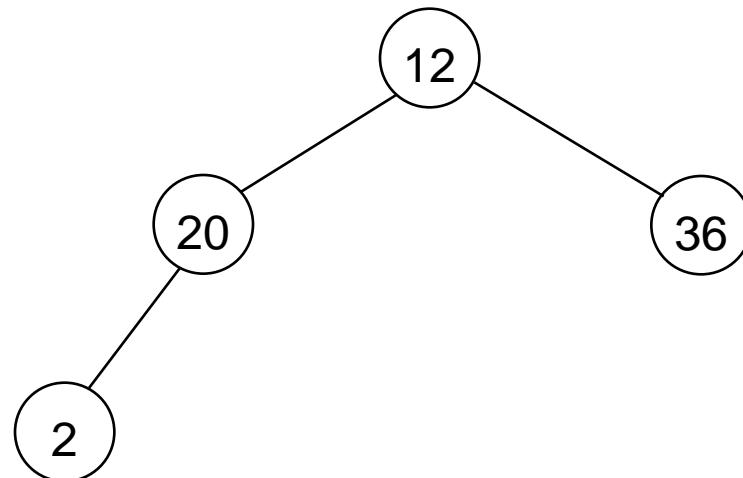


MONCEAU

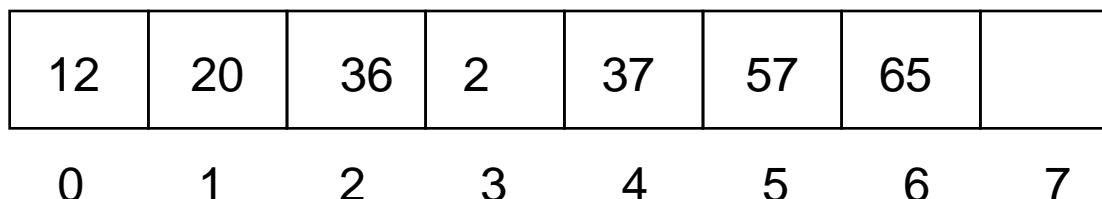


# Tri par monceau - Exemple

*Retrait du 3<sup>ème</sup> élément*

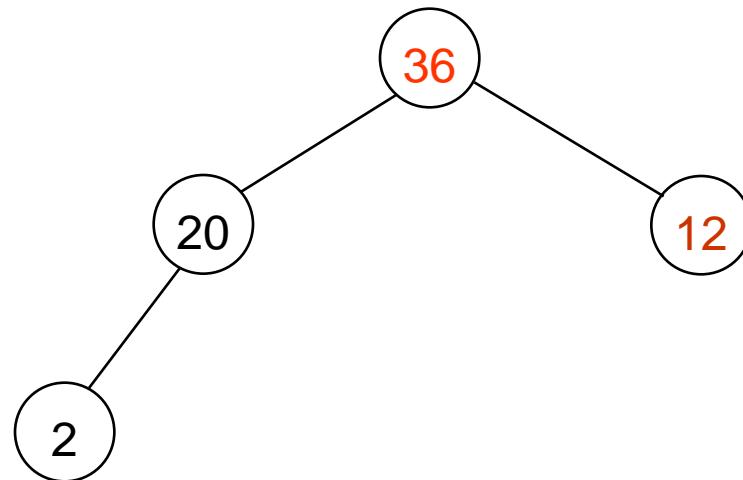


MONCEAU

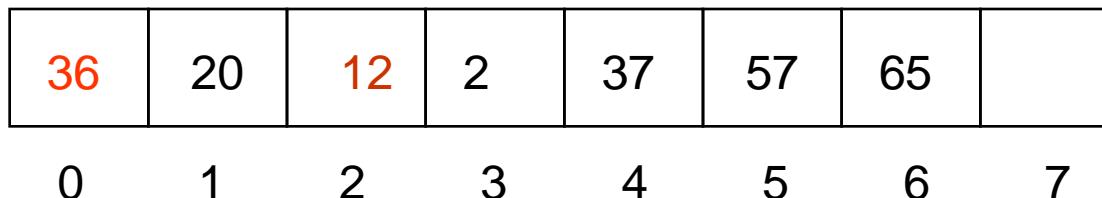


# Tri par monceau - Exemple

*Percoler*

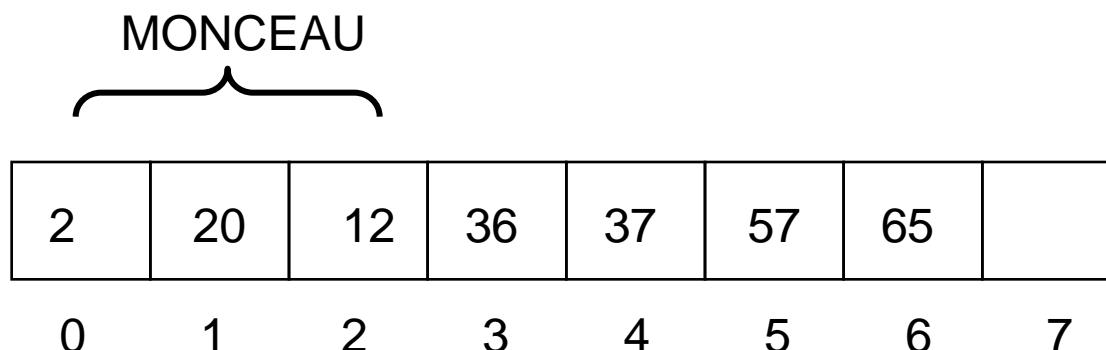
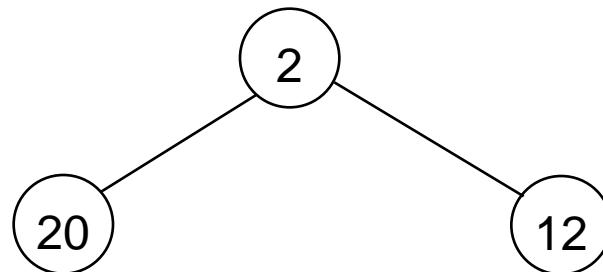


MONCEAU



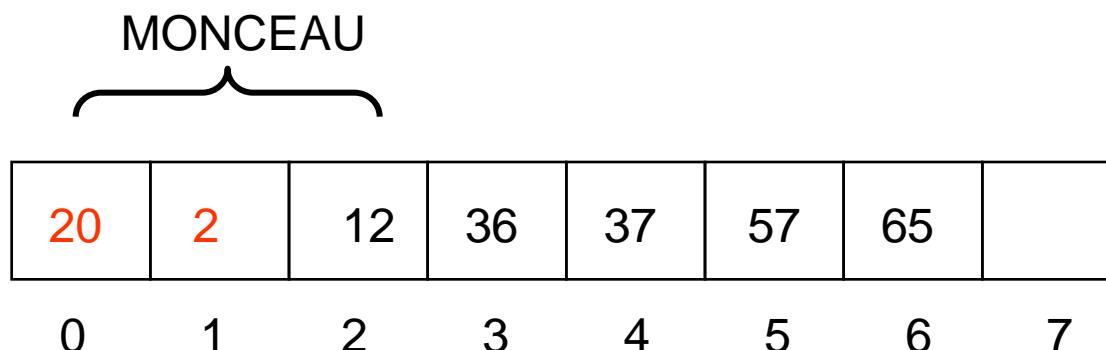
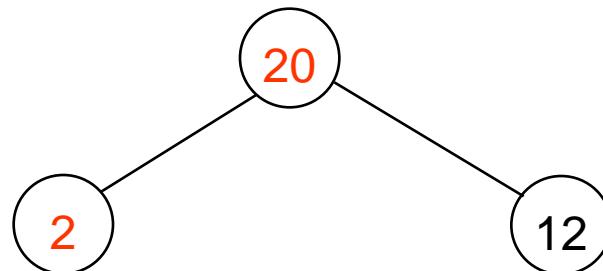
# Tri par monceau - Exemple

*Retrait du 4<sup>ème</sup> élément*



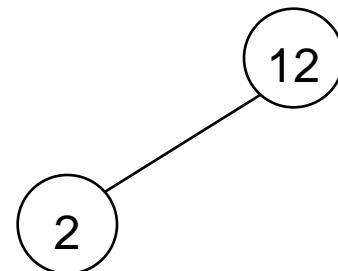
# Tri par monceau - Exemple

*Percoler*



# Tri par monceau - Exemple

*Retrait du 5<sup>ème</sup> élément*



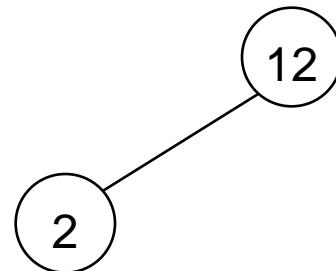
MONCEAU



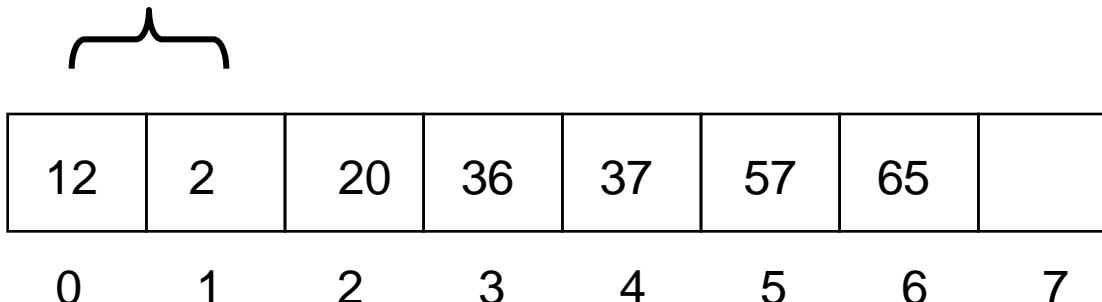
12	2	20	36	37	57	65	
0	1	2	3	4	5	6	7

# Tri par monceau - Exemple

*Percoler*



MONCEAU



# Tri par monceau - Exemple

*Retrait du 6<sup>ème</sup> élément*

2

Le tableau est maintenant trié

2	12	20	36	37	57	65	
0	1	2	3	4	5	6	7

# Tri par monceau

```
/**  
 * Standard heapsort.  
 * @param a an array of Comparable items.  
 */  
public static <AnyType extends Comparable<? super AnyType>>  
void heapsort( AnyType [ ] a )  
{  
    for( int i = a.length / 2; i >= 0; i-- ) /* construire le monceau */  
        percDown( a, i, a.length );  
    for( int i = a.length - 1; i > 0; i-- )  
    {  
        swapReferences( a, 0, i ); /* permuter le maximum (racine)  
                               avec le dernière élément du monceau */  
        percDown( a, 0, i );  
    }  
}
```

# Tri par monceau (2)

```
/**  
 * Internal method for heapsort that is used in deleteMax and buildHeap.  
 * @param a: un tableau dont les éléments sont de type Comparable.  
 * @int i: la position de l'élément à percoler.  
 * @int n: la position du dernière élément du monceau.  
 */  
private static <AnyType extends Comparable<? super AnyType>>  
void percDown( AnyType [ ] a, int i, int n ) {  
    int child;  
    AnyType tmp;  
    for( tmp = a[ i ]; leftChild( i ) < n; i = child ) {  
        child = leftChild( i );  
        if( child != n - 1 && a[ child ].compareTo( a[ child + 1 ] ) < 0 )  
            child++;  
        if( tmp.compareTo( a[ child ] ) < 0 )  
            a[ i ] = a[ child ];  
        else  
            break;  
    }  
    a[ i ] = tmp;  
}
```

# Tri par monceau (3)

```
/**  
 * Internal method for heapsort.  
 * @param i the index of an item in the heap.  
 * @return the index of the left child.  
 */  
private static int leftChild( int i )  
{  
    return 2 * i + 1;  
}
```

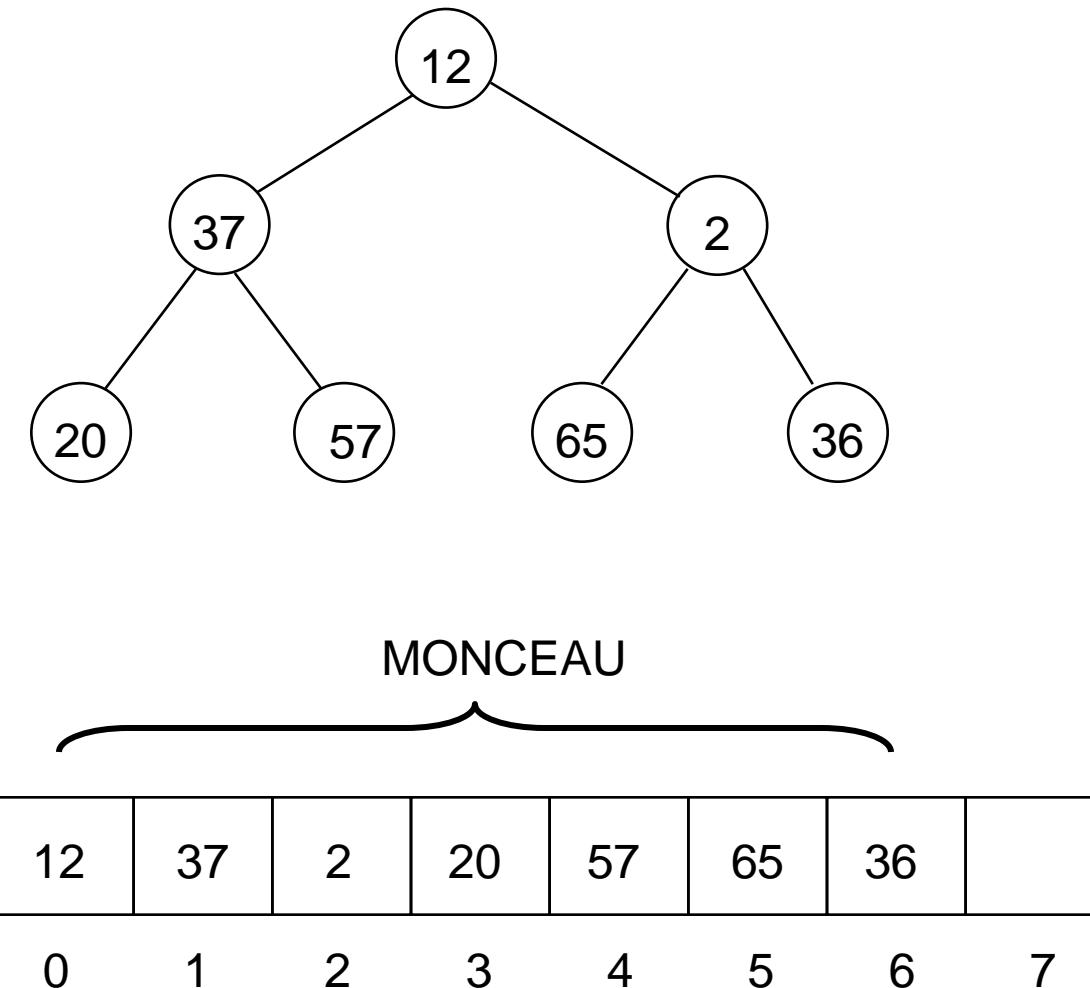
# Tri partiel

- Supposons maintenant un monceau dont la racine est le plus petit élément (contrairement à l'exemple précédent)
- Supposons que le tableau contient  $n$  éléments, et une valeur  $m < n$
- On remarquera que après  $m$  itérations, on obtient, à la fin du tableau, et en ordre inverse, les  $m$  plus petits éléments du tableau

# Tri partiel (2)

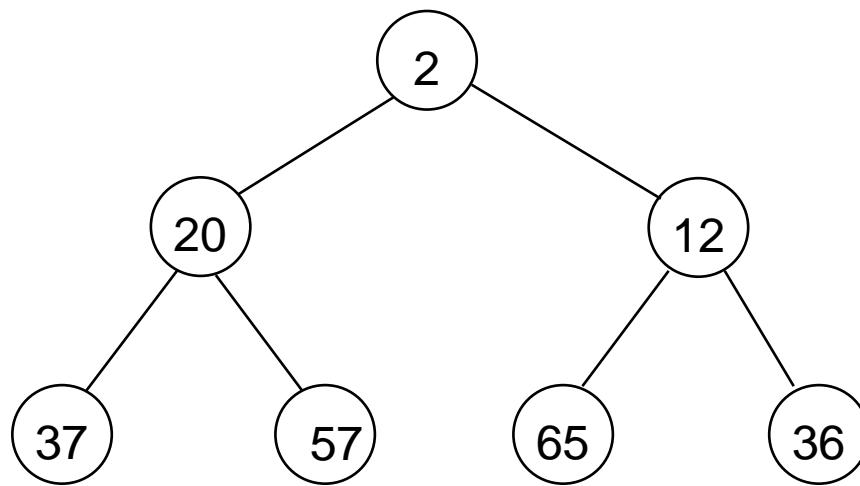
- On peut donc, de manière efficace, trier les  $m$  plus petits éléments du tableau:  $O(m \lg n)$
- C'est le même principe que celui du tri par sélection, mais le tri par sélection est moins efficace:  $O(mn)$

# Tri partiel - exemple

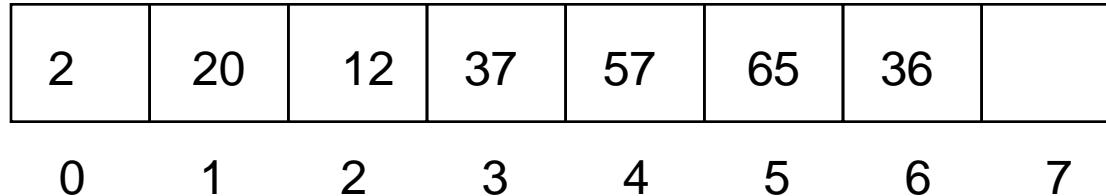


# Tri partiel - exemple

*Création du monceau*

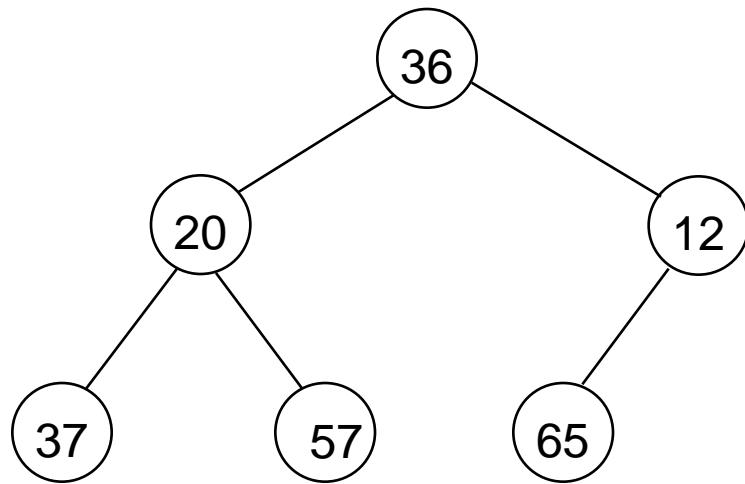


MONCEAU

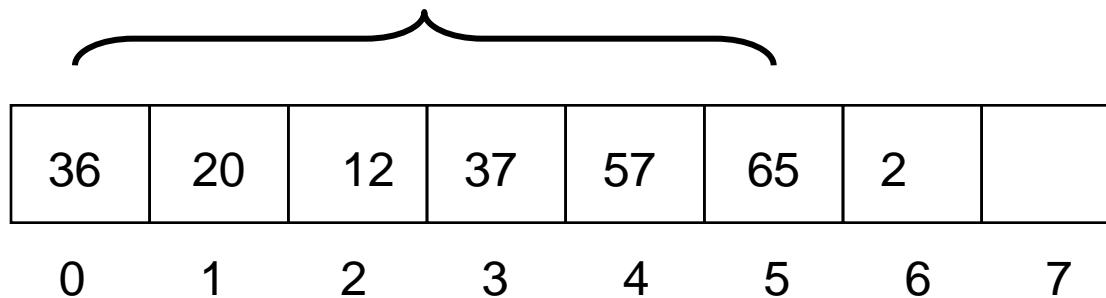


# Tri partiel - exemple

*Retrait du 1<sup>er</sup> élément*

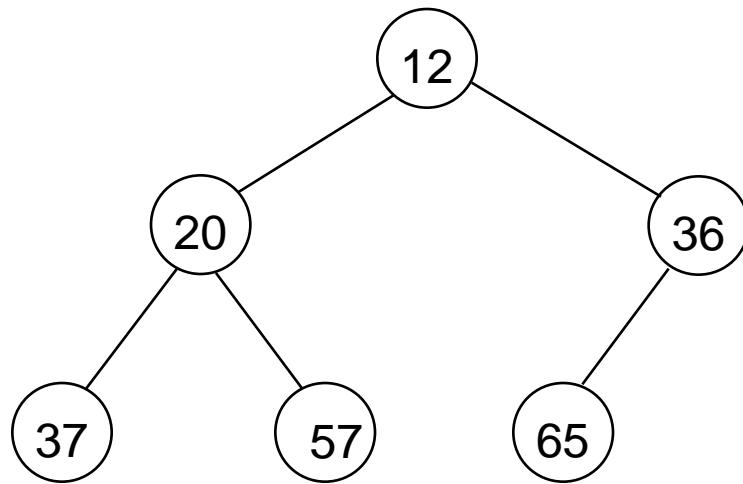


MONCEAU

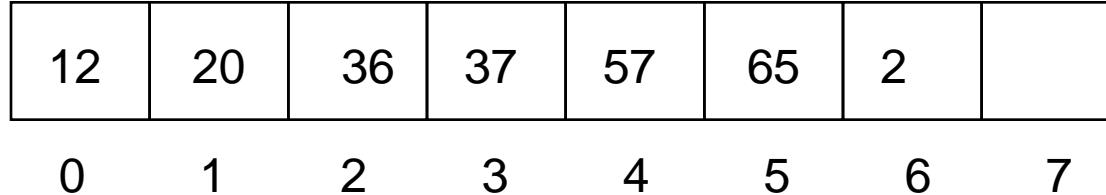


# Tri partiel - exemple

*Percoler*

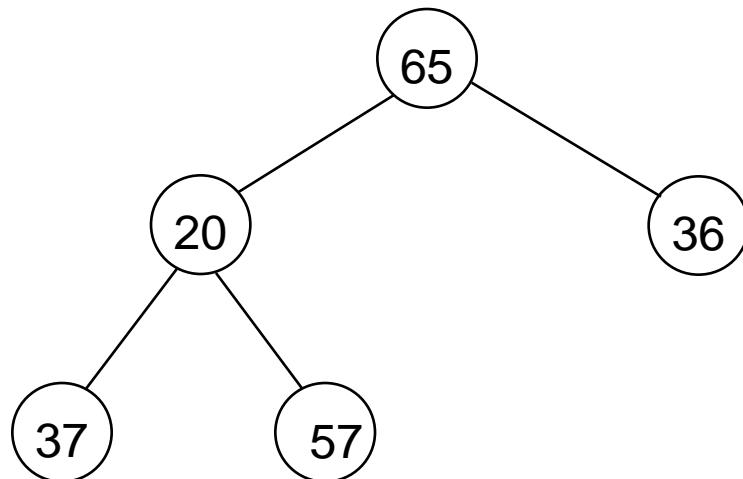


MONCEAU

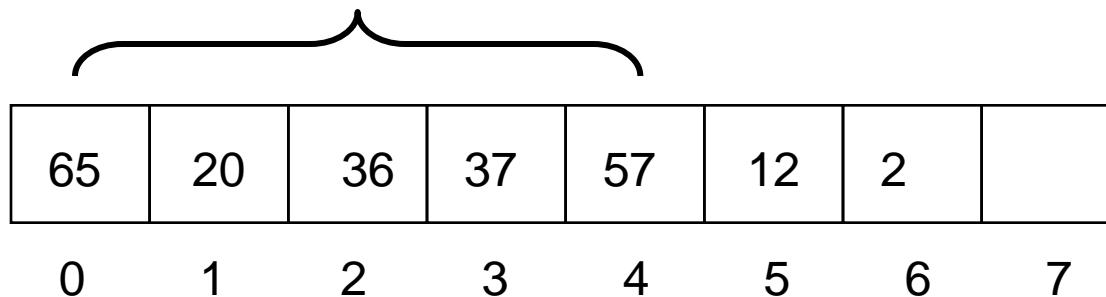


# Tri partiel - exemple

*Retrait du 2<sup>ème</sup> élément*

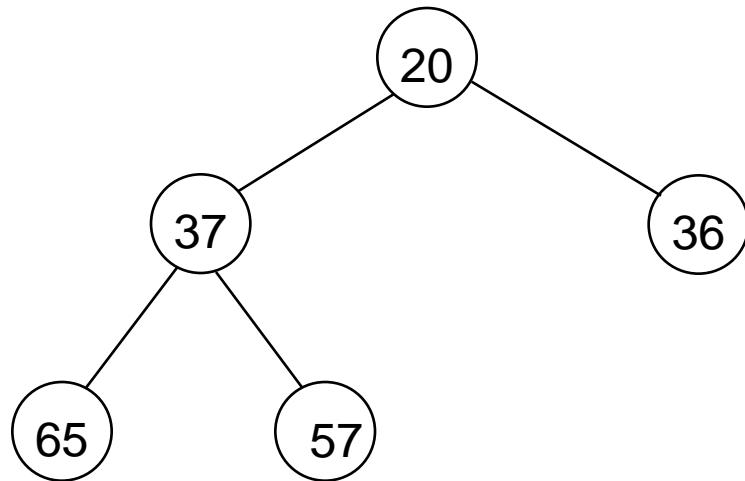


MONCEAU

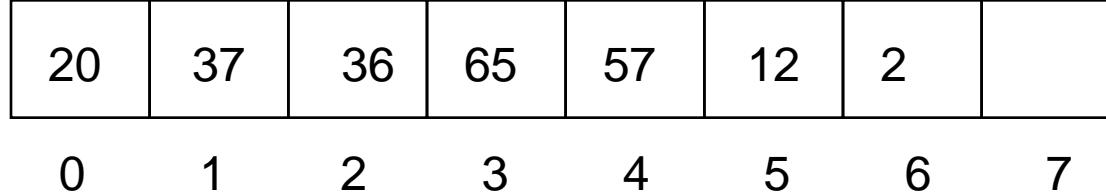


# Tri partiel - exemple

*Percoler*

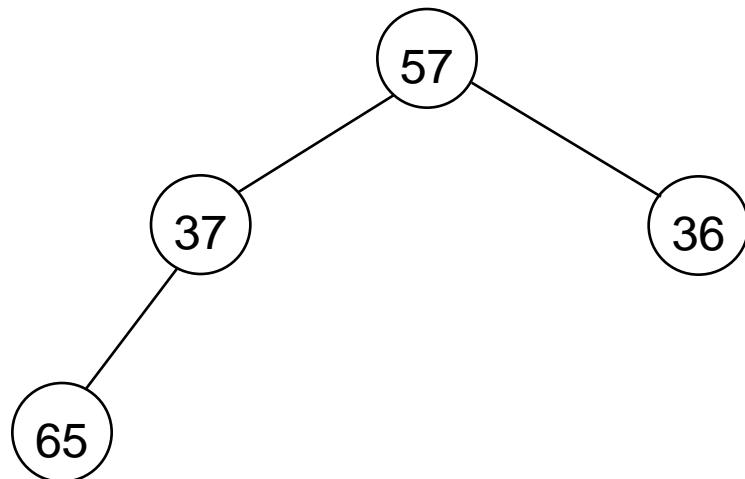


MONCEAU

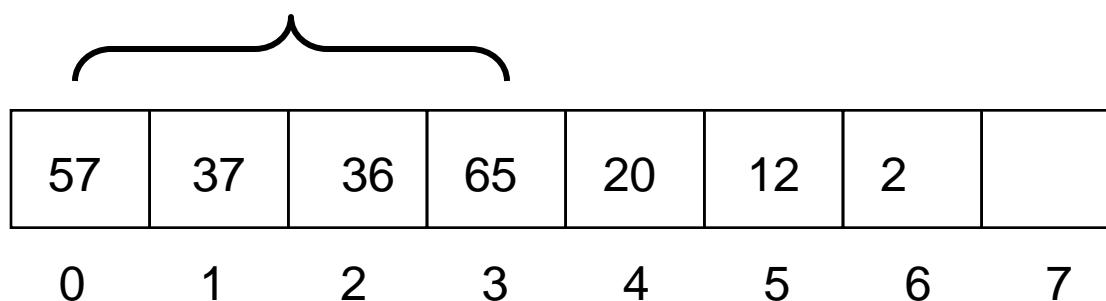


# Tri partiel - exemple

*Retrait du 3<sup>ème</sup> élément*

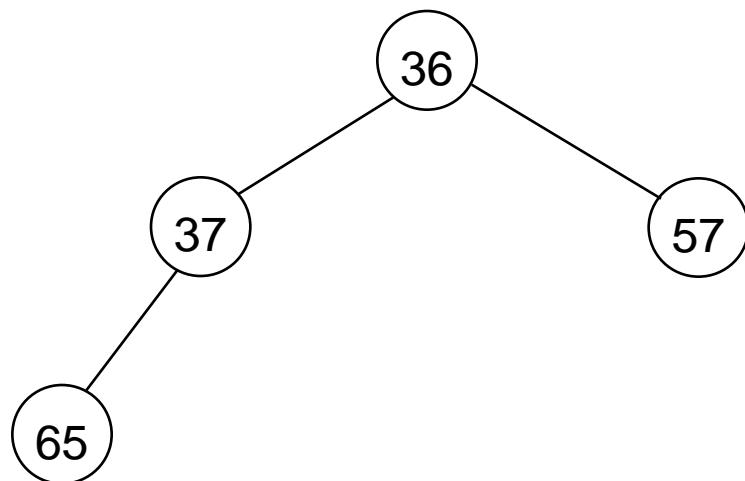


MONCEAU

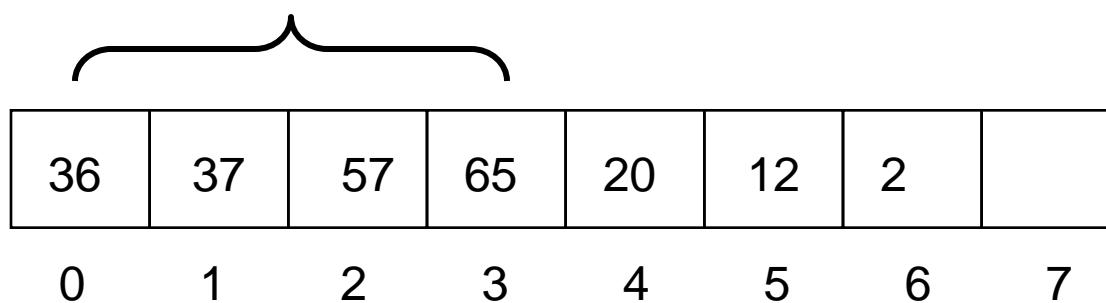


# Tri partiel - exemple

*Percoler*

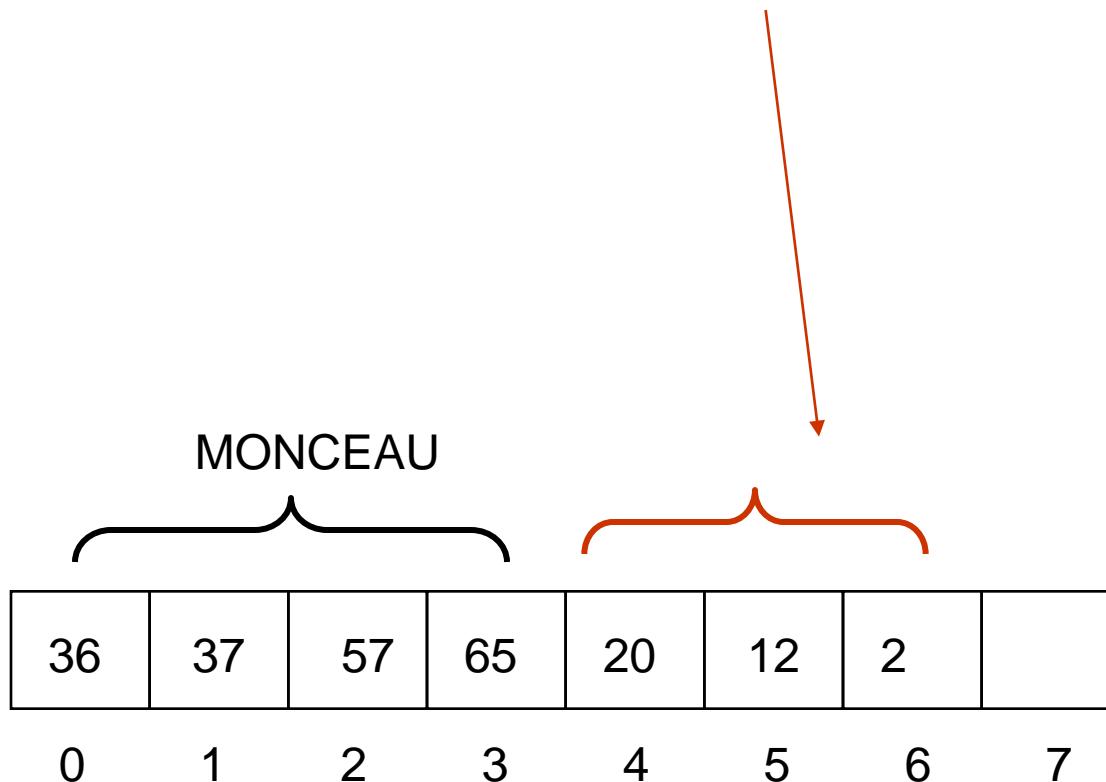


MONCEAU



# Tri partiel - exemple

Si on arrête ici, on voit qu'on a bien obtenu les 3 plus petits éléments, qui sont en ordre croissant si on parcourt le tableau à partir de la fin



# Exercice

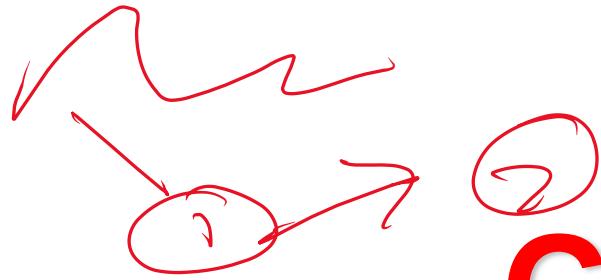
On veut ajouter une fonction **int findMax( )** à la classe BinaryHeap.  
Donnez le code de la fonction **int findMax( )** en assurant un temps  
d'exécution dans le pire cas qui ne doit pas dépasser `currentSize/2`.

# Solution

```
AnyType findMax( )
{
    AnyType max = array[1];

    for( int i = currentSize/2; i <= currentSize; i++ )
    {
        if (array[i].compareTo (max) >0)
            max = array[i];
    }

    return max ;
}
```



# Graphes

# Graphes

---

- 1.Définitions et exemples
  - 2.Implémentations
  - 3.Ordre topologique
  - 4.Chemin le plus court
  - 5.Dijkstra
  - 6.Parcours
-

# Graphes

---

1. Définitions et exemples
2. Implémentations
3. Ordre topologique
4. Chemin le plus court
5. Dijkstra
6. Parcours

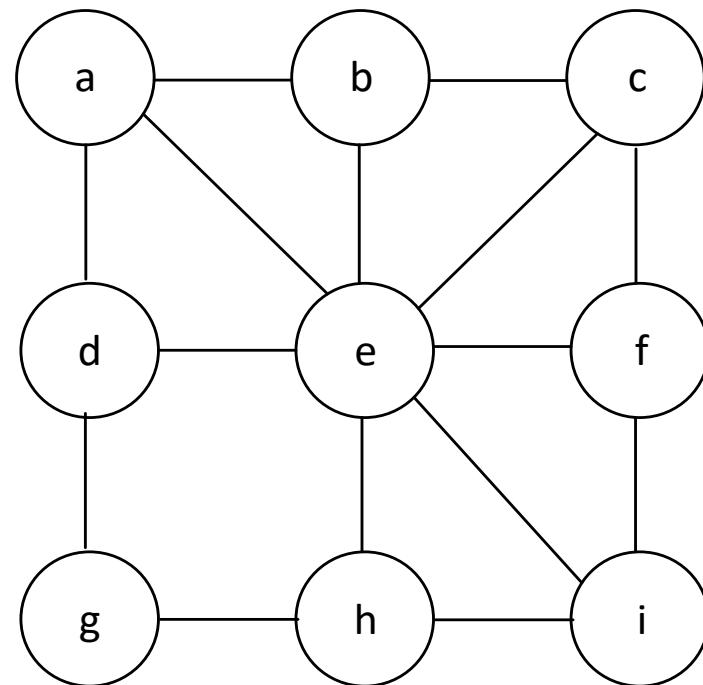
# Graphes - définitions

---

- Un graphe est une paire  $G = (V, E)$  où  $V$  est un ensemble de **sommets** et  $E$  un ensemble d'**arêtes**. Chaque arête est une paire qui relie deux sommets du graphe.

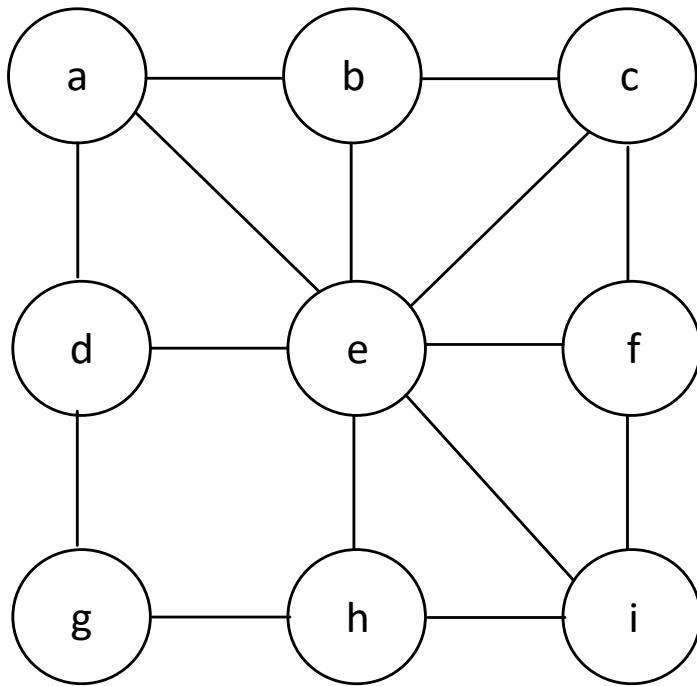
$$V = \{a, b, c, d, e, f, g, h, i\}$$

$$E = \{(a, b), (a, d), (a, e), (b, c), (b, d), (c, e), (c, f), (d, e), (d, g), (e, f), (e, h), (e, i), (f, i), (g, h), (h, i)\}$$

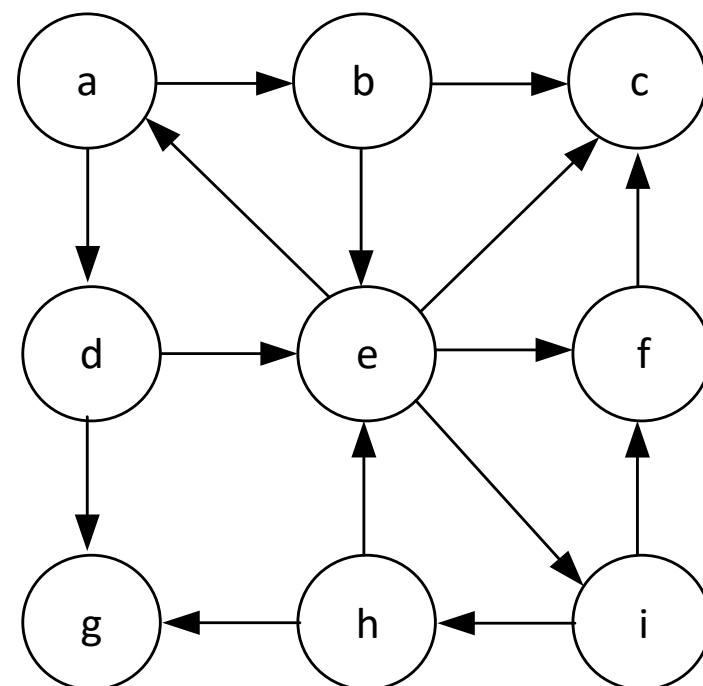


# Graphes - définitions

- Graphe orienté: les sommets sont reliés par des **arcs** (arêtes orientées), qui relient un sommet origine à un sommet destination



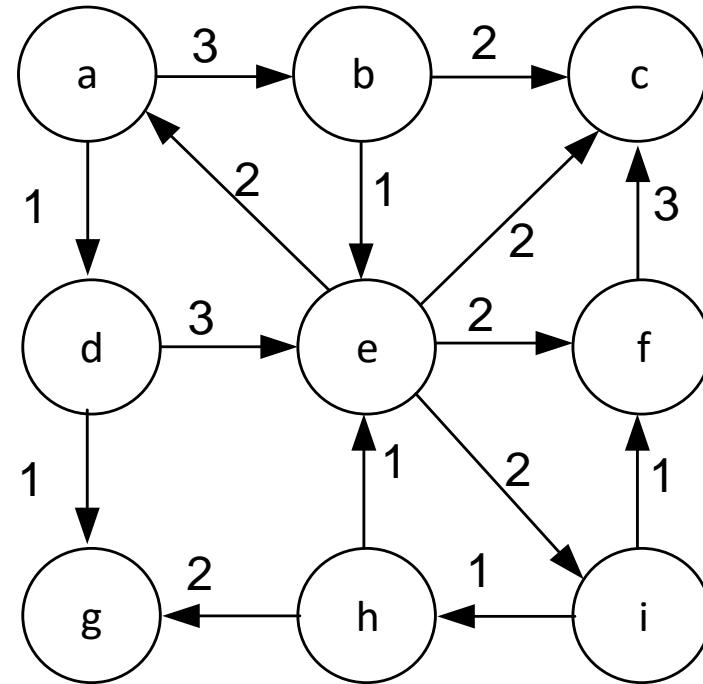
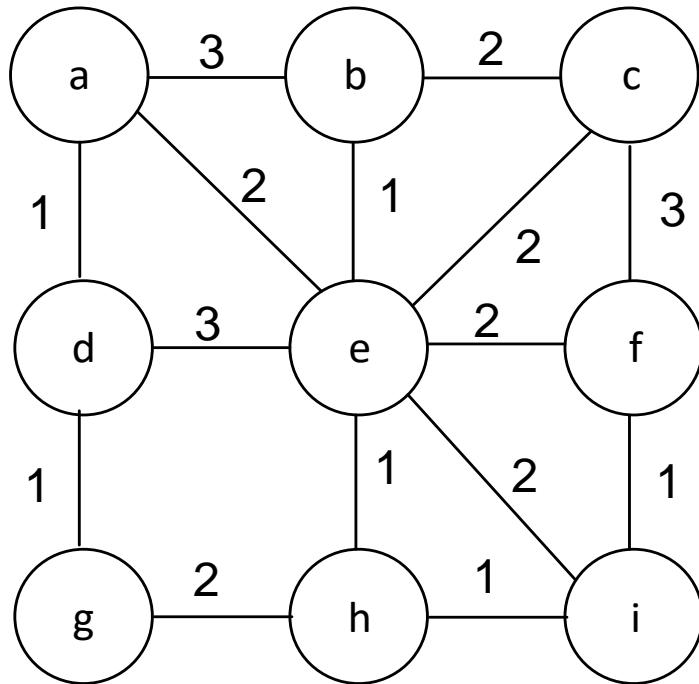
Graphe non-orienté



Graphe orienté

# Graphes - définitions

- Un graphe est dit **valué** si les arêtes (ou arcs) ont une valeur indiquant le **coût** pour les traverser. On peut aussi parler de **poids** de chaque arête (arc).



# Graphes – définitions

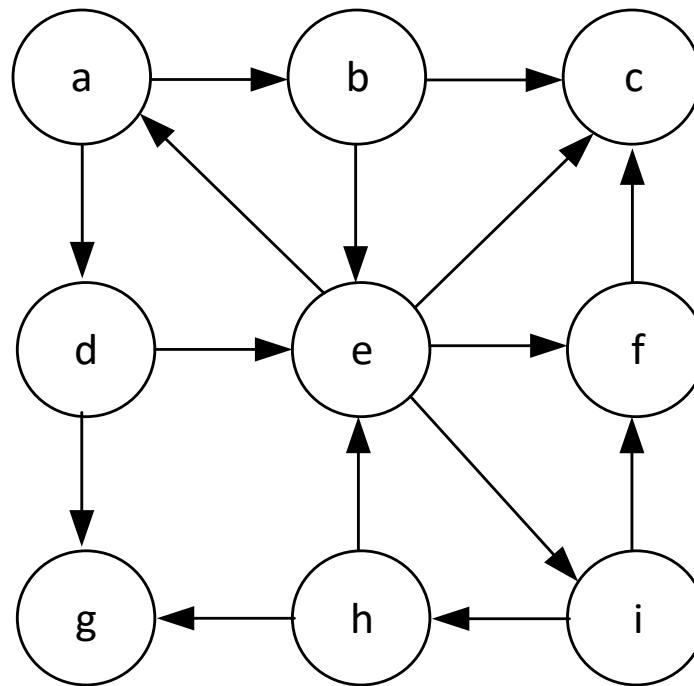
---

- Un **chemin** est une séquence de sommets du graphe connectés par des arêtes
  - La **longueur** d'un chemin correspond au nombre d'arêtes dans ce chemin
  - Un **chemin simple** ne contient pas plus d'une fois le même sommet
  - Un **cycle** est un chemin qui commence et termine au même sommet
  - Un **graphe orienté acyclique** est un graphe orienté qui ne contient pas de cycle
-

# Graphes – définitions

---

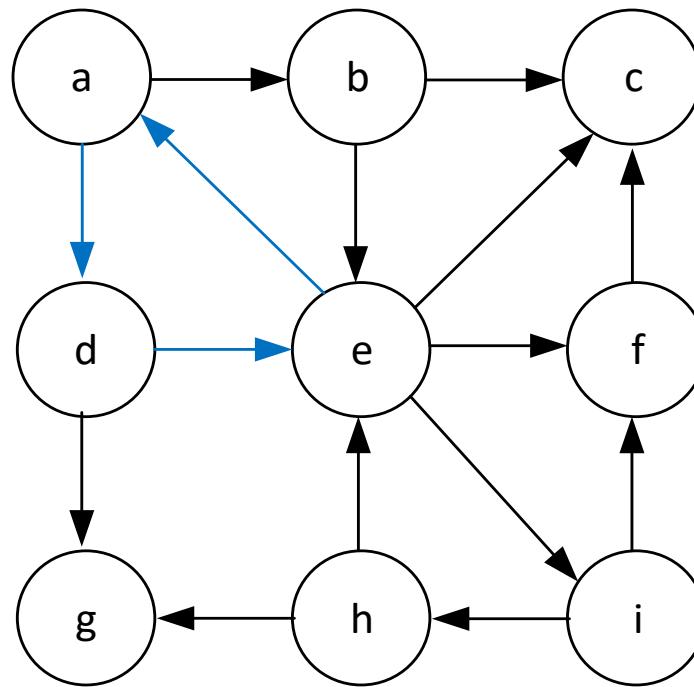
- Est-ce un graphe orienté acyclique ?



# Graphes – définitions

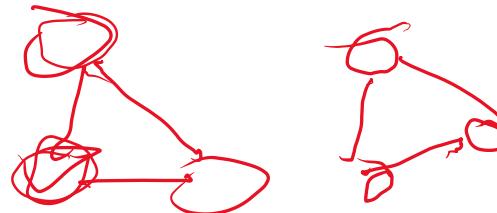
---

- Est-ce un graphe orienté acyclique ? **NON**



# Graphes – définitions

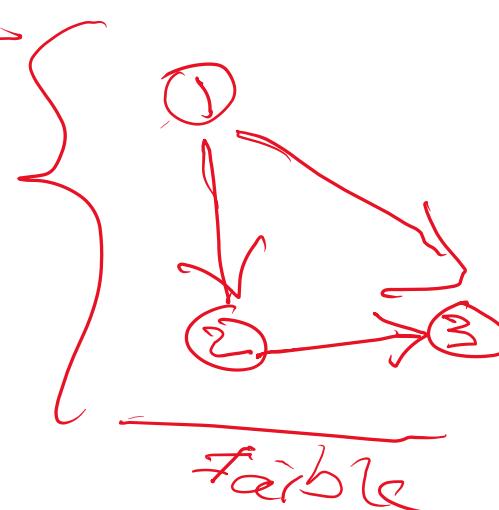
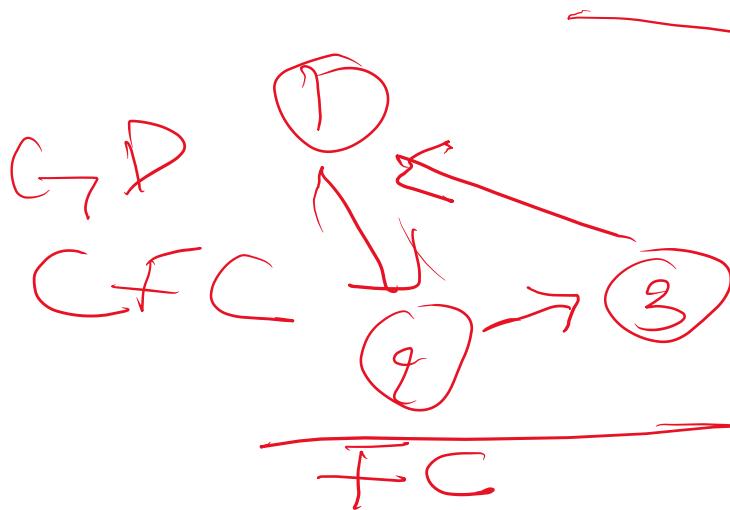
- Graphe connexe → un chemin pour chaque paire de nœuds



- Graphes orientés

- connexes → connexité forte

- Non connexes, mais le graphe sous-jacent sans orientation est connexe → connexité faible



1	2
1	3
2	1
2	3
3	1
3	2

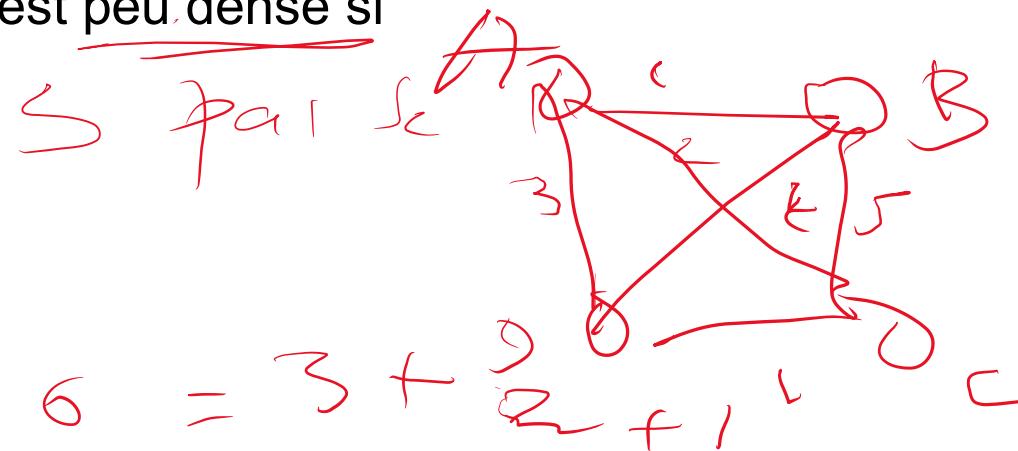


# Graphes – définitions

- Un graphe complet comportant  $|V|$  noeuds possède  $|E| = (\underline{|V|} - 1) \cdot (\underline{|V|}) / 2$  arcs
  - On dit qu'un graphe est dense si  $|E|$  est  $\Theta(|V|^2)$
  - On dira qu'un graphe est peu dense si  $|E|$  est  $\Theta(|V|)$



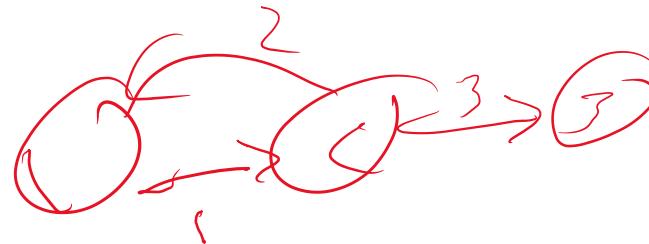
$$\left(\frac{N}{2} \left(N - 1\right)\right)_6 = 3 + \overbrace{2 + 1}^6$$



# Graphes – exemples

---

- Réseaux
  - Sociaux
  - Ordinateurs
  - Transports
- Théorie des langages
  - Automates
  - Graphe de flot de contrôle
  - Graphes d'appel
  - Graphes des dépendances



# Graphes – exemples

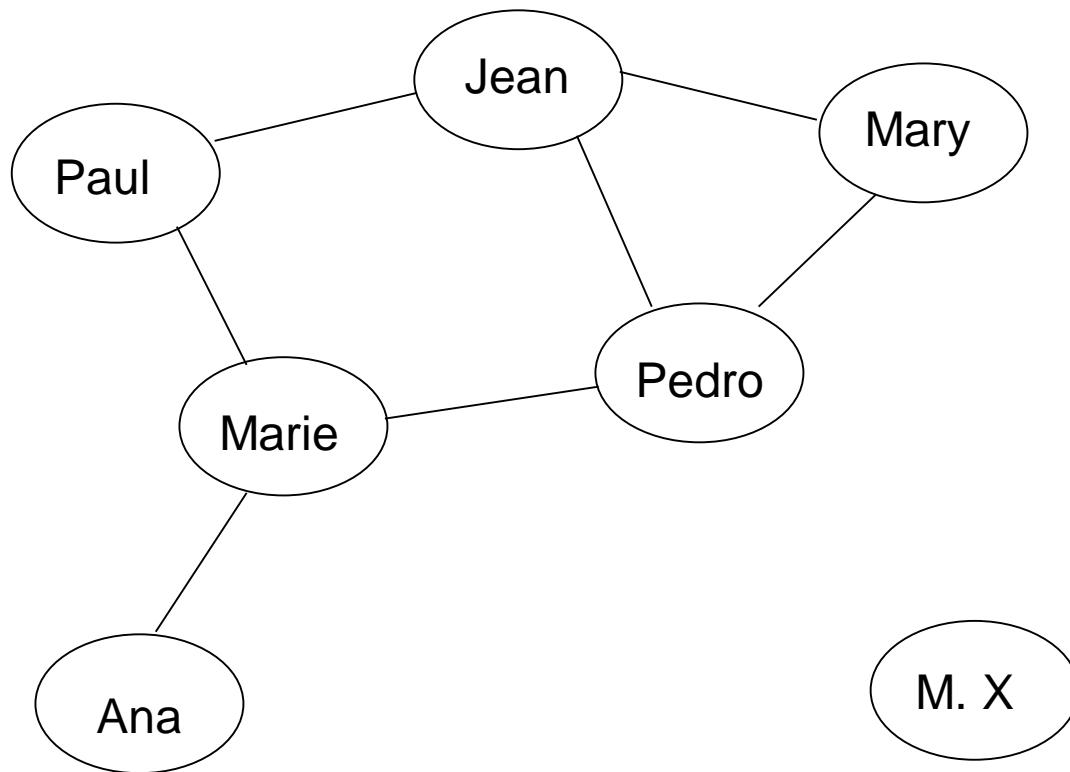
---

- Généalogies
- Biologie moléculaire
  - Chaînes métaboliques
  - Représentation de protéines et de molécules organiques
- Génie logiciel
  - Diagrammes UML (classe, interaction)
  - Diagramme de transition des Interfaces usager

# Graphes - exemples

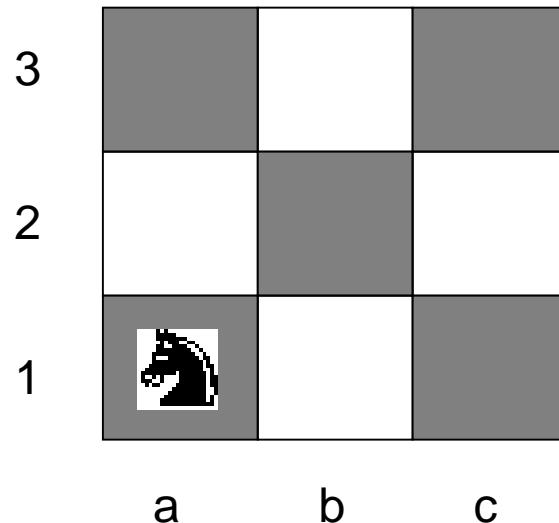
---

Graphe non orienté: chaque arête représente deux personnes qui se connaissent



# Graphes – exemples

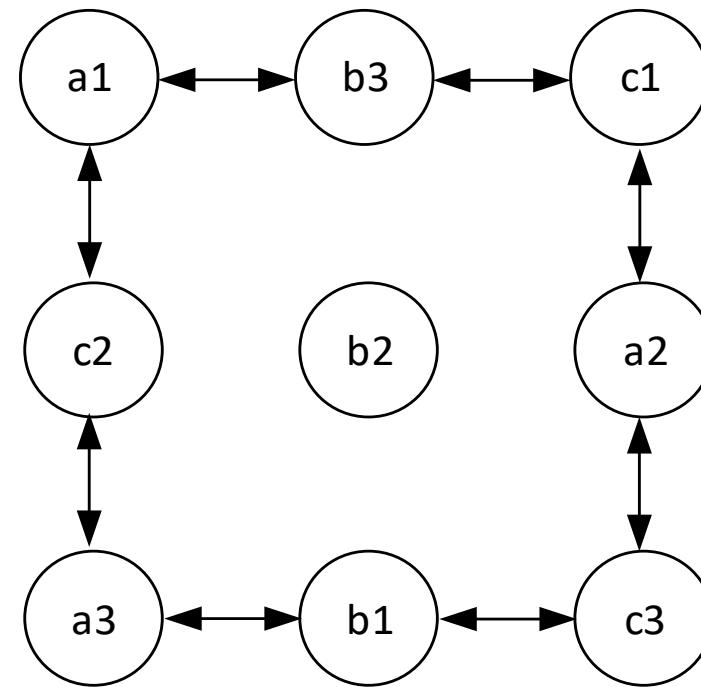
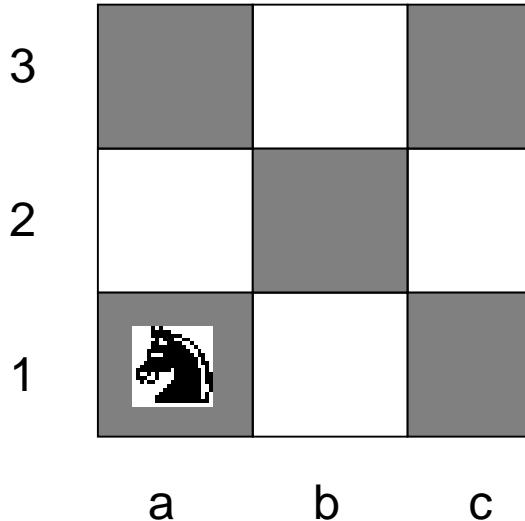
---



Quelles sont les positions possibles du cavalier à partir de sa position actuelle?

# Graphes – exemples

---



Remarque: dans la figure, chaque arc représente en fait deux arcs,  
soit un pour chaque orientation

---

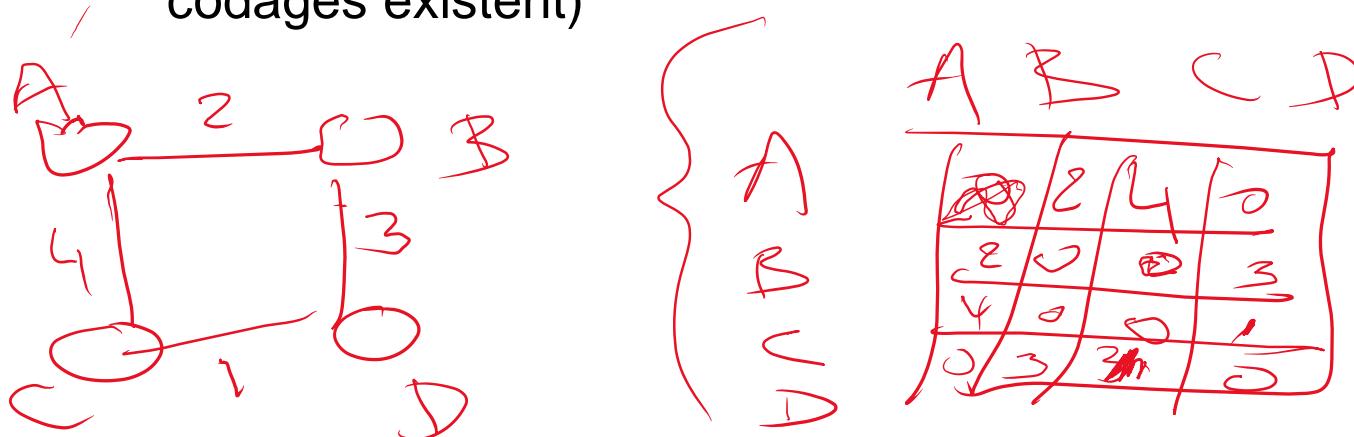
# Graphes

---

- 1.Définitions et exemples
- 2.Implémentations**
- 3.Ordre topologique
- 4.Chemin le plus court
- 5.Dijkstra
- 6.Parcours

# Graphes - implémentation

- Matrice d'adjacence
  - On suppose que les sommets du graphe sont étiquetés de 0 à N
  - S'il existe une arête du sommet  $i$  au sommet  $j$ , on met 1 à la position  $A[i][j]$ , sinon on met INFINI comme valeur (autres codages existent)



# Graphes – implémentation

---

- Matrice d'adjacence
  - Si le graphe est valué, on met à la position  $A[i][j]$ , le poids associé à l'arête
  - Si le graphe est peu dense, ce qui est souvent le cas, il y aura beaucoup de 0 dans la matrice

# Graphes – implémentation

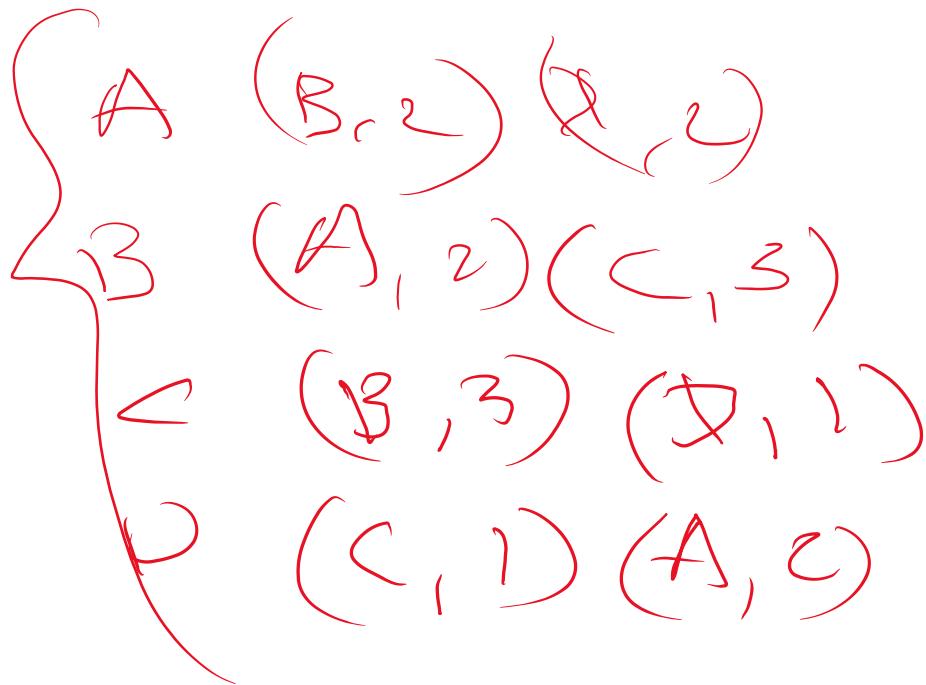
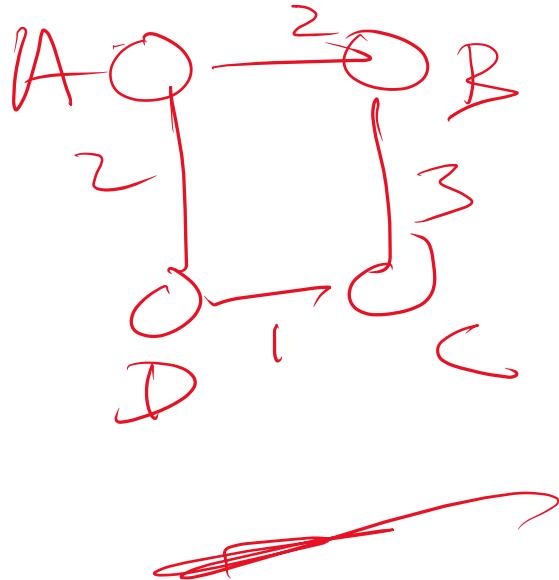
---

- Listes d'adjacence
  - Pour chaque sommet, on associe une liste de tous les autres sommets auquel il est lié par une arête dont il est l'origine
  - En principe (tout comme avec la matrice d'adjacence), il faut une table qui associe l'identificateur de chaque sommet à un numéro interne dans la représentation
  - En Java, cette table peut nous retourner une référence sur la structure qui représente le sommet

# Graphes – implémentation

---

- Listes d'adjacence



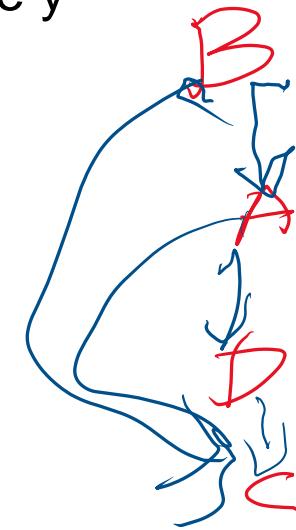
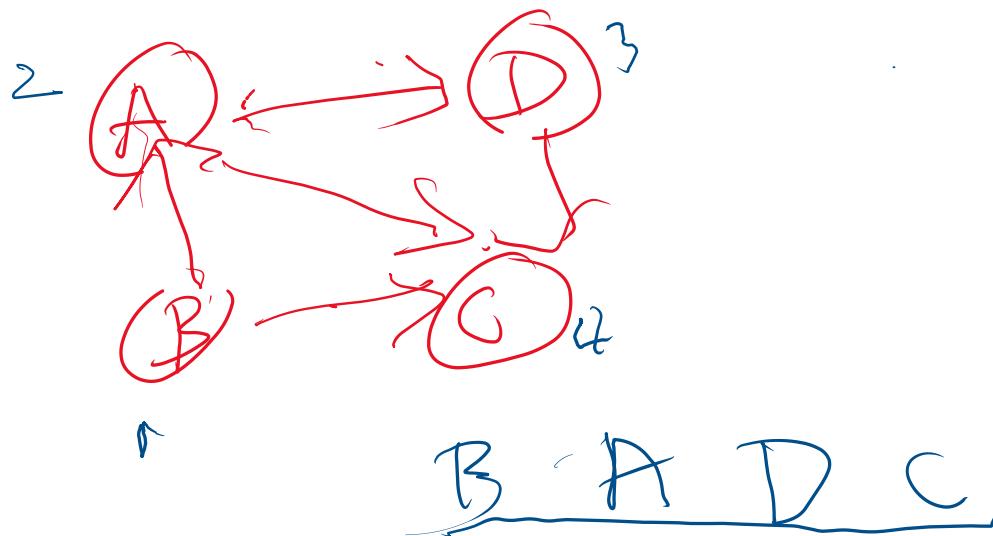
# Graphes

---

- 1.Définitions et exemples
- 2.Implémentations
- 3.Ordre topologique**
- 4.Chemin le plus court
- 5.Dijkstra
- 6.Parcours

# Ordre topologique

- Graphes orientés acycliques
- Définition
  - Ordre sur les nœuds du graphe dans lequel l'existence d'un chemin entre  $x$  et  $y$  implique que  $x$  précède  $y$



# Algorithme

---

```
void topsort( ) throws CycleFoundException
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == null )
            throw new CycleFoundException( );
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

# Algorithme

---

Problème?

```
void topsort( ) throws CycleFoundException
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == null )
            throw new CycleFoundException( );
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

# Algorithme

---

```
void topsort( ) throws CycleFoundException
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == null )
            throw new CycleFoundException( );
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

Complexité:  $O(|V|^2)$

---

# Algorithme amélioré (?)

---

```
void topsort( ) throws CycleFoundException
{
    Queue<Vertex> q = new Queue<Vertex>( );
    int counter = 0;

    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter; // Assign next number

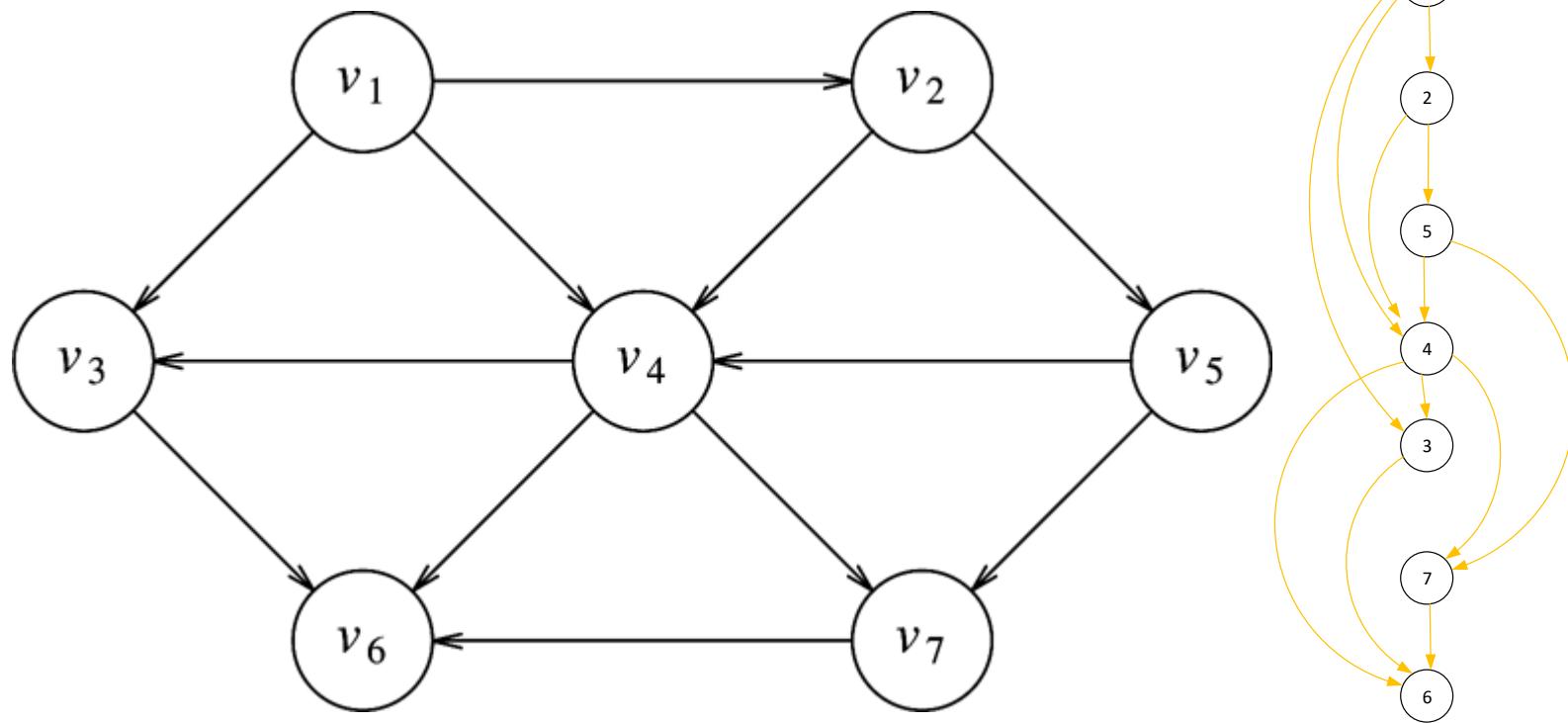
        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }

    if( counter != NUM_VERTICES )
        throw new CycleFoundException( );
}
```

- Complexité:  $O(|E| + |V|)$
- Algorithme avec une file (liste de travail)

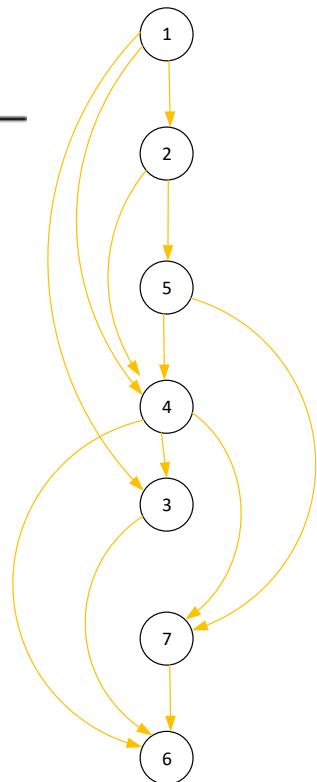
# Exemple

---



# Simulation

Vertex	Indegree Before Dequeue #						
	1	2	3	4	5	6	7
$v_1$	0	0	0	0	0	0	0
$v_2$	1	0	0	0	0	0	0
$v_3$	2	1	1	1	0	0	0
$v_4$	3	2	1	0	0	0	0
$v_5$	1	1	0	0	0	0	0
$v_6$	3	3	3	3	2	1	0
$v_7$	2	2	2	1	0	0	0
<i>Enqueue</i>	$v_1$	$v_2$	$v_5$	$v_4$	$v_3, v_7$		$v_6$
<i>Dequeue</i>	$v_1$	$v_2$	$v_5$	$v_4$	$v_3$	$v_7$	$v_6$



# Graphes

---

- 1.Définitions et exemples
- 2.Implémentations
- 3.Ordre topologique
- 4.Chemin le plus court**
- 5.Dijkstra
- 6.Parcours

# Plus court chemin sans poids

---

- Graphe orienté
- Nœud de départ
- Coût associé aux arêtes
  - Longueur du chemin

# Algorithme

---

```
void unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

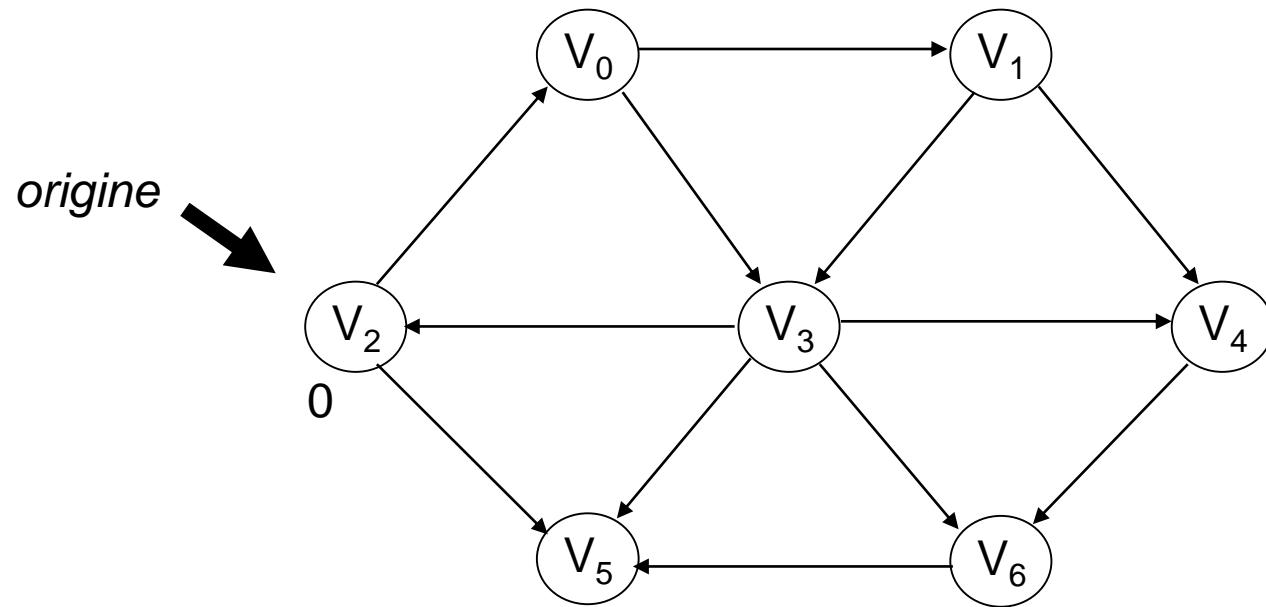
    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

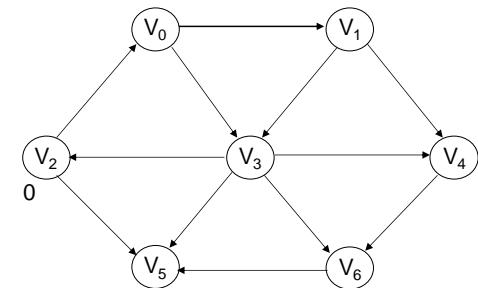
- Complexité:  $O(|V|^2)$

# Exemple

---



# Simulation



Nœuds	Distance	Connu?	Parent
V <sub>0</sub>	$\infty$	Faux	-
V <sub>1</sub>	$\infty$	Faux	-
V <sub>2</sub>	$\infty$	Faux	-
V <sub>3</sub>	$\infty$	Faux	-
V <sub>4</sub>	$\infty$	Faux	-
V <sub>5</sub>	$\infty$	Faux	-
V <sub>6</sub>	$\infty$	Faux	-



Nœuds	Distance	Connu?	Parent
V <sub>0</sub>	1	Faux	V <sub>2</sub>
V <sub>1</sub>	$\infty$	Faux	-
V <sub>2</sub>	0	Vrai	-
V <sub>3</sub>	$\infty$	Faux	-
V <sub>4</sub>	$\infty$	Faux	-
V <sub>5</sub>	1	Faux	V <sub>2</sub>
V <sub>6</sub>	$\infty$	Faux	-

Nœuds	Distance	Connu?	Parent
V <sub>0</sub>	1	Vrai	V <sub>2</sub>
V <sub>1</sub>	2	Faux	V <sub>0</sub>
V <sub>2</sub>	0	Vrai	-
V <sub>3</sub>	2	Faux	V <sub>0</sub>
V <sub>4</sub>	$\infty$	Faux	-
V <sub>5</sub>	1	Faux	V <sub>2</sub>
V <sub>6</sub>	$\infty$	Faux	-



Nœuds	Distance	Connu?	Parent
V <sub>0</sub>	1	Vrai	V <sub>2</sub>
V <sub>1</sub>	2	Faux	V <sub>0</sub>
V <sub>2</sub>	0	Vrai	-
V <sub>3</sub>	2	Faux	V <sub>0</sub>
V <sub>4</sub>	$\infty$	Faux	-
V <sub>5</sub>	1	Vrai	V <sub>2</sub>
V <sub>6</sub>	$\infty$	Faux	-

# Simulation

Nœuds	Distance	Connu?	Parent
V <sub>0</sub>	1	Vrai	V <sub>2</sub>
<b>V<sub>1</sub></b>	<b>2</b>	<b>Vrai</b>	<b>V<sub>0</sub></b>
V <sub>2</sub>	0	Vrai	-
V <sub>3</sub>	2	Faux	V <sub>0</sub>
V <sub>4</sub>	3	Faux	V <sub>1</sub>
V <sub>5</sub>	1	Vrai	V <sub>2</sub>
V <sub>6</sub>	$\infty$	Faux	-



Nœuds	Distance	Connu?	Parent
V <sub>0</sub>	1	Vrai	V <sub>2</sub>
V <sub>1</sub>	2	Vrai	V <sub>0</sub>
V <sub>2</sub>	0	Vrai	-
<b>V<sub>3</sub></b>	<b>2</b>	<b>Vrai</b>	<b>V<sub>0</sub></b>
V <sub>4</sub>	3	Faux	V <sub>1</sub>
V <sub>5</sub>	1	Vrai	V <sub>2</sub>
V <sub>6</sub>	3	Faux	V <sub>3</sub>

Nœuds	Distance	Connu?	Parent
V <sub>0</sub>	1	Vrai	V <sub>2</sub>
V <sub>1</sub>	2	Vrai	V <sub>0</sub>
V <sub>2</sub>	0	Vrai	-
V <sub>3</sub>	2	Vrai	V <sub>0</sub>
<b>V<sub>4</sub></b>	<b>3</b>	<b>Vrai</b>	<b>V<sub>1</sub></b>
V <sub>5</sub>	1	Vrai	V <sub>2</sub>
V <sub>6</sub>	3	Faux	V <sub>3</sub>

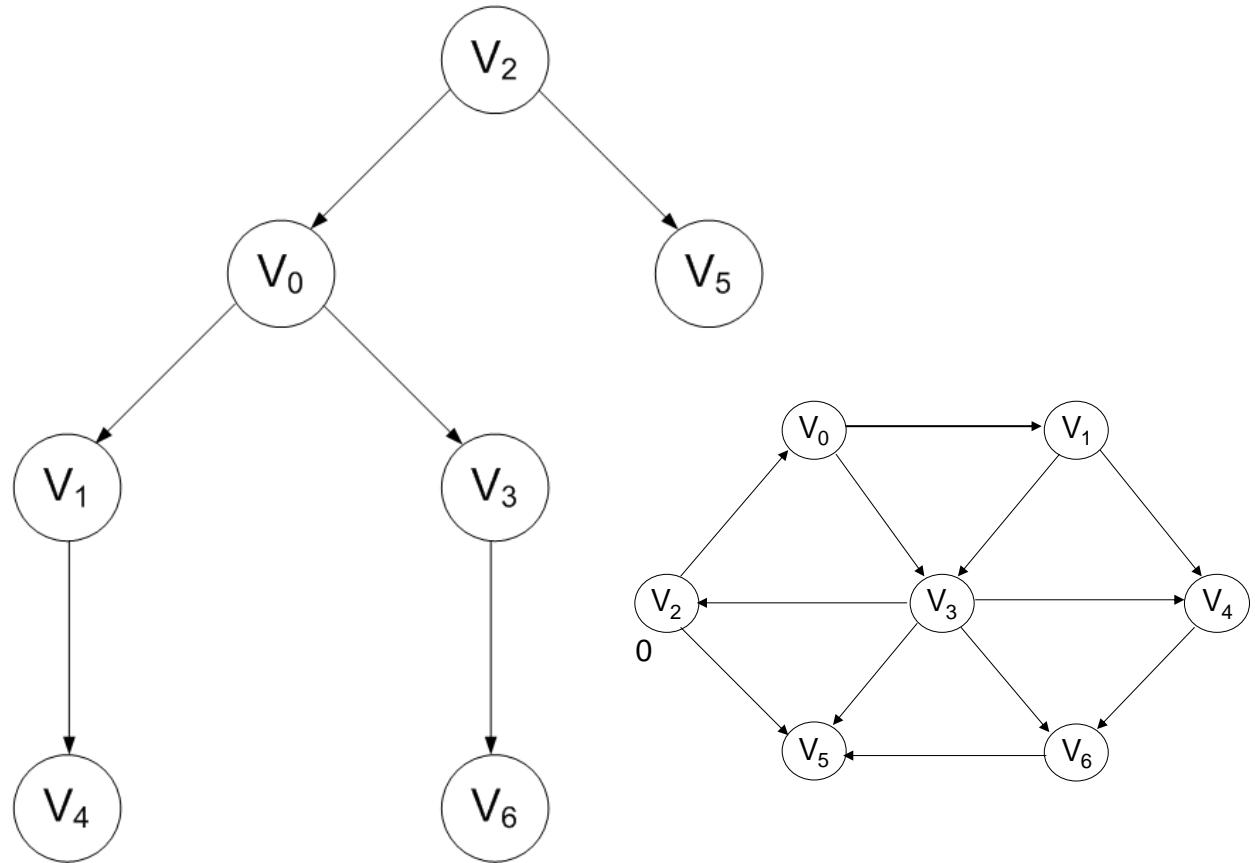


Nœuds	Distance	Connu?	Parent
V <sub>0</sub>	1	Vrai	V <sub>2</sub>
V <sub>1</sub>	2	Vrai	V <sub>0</sub>
V <sub>2</sub>	0	Vrai	-
V <sub>3</sub>	2	Vrai	V <sub>0</sub>
V <sub>4</sub>	3	Vrai	V <sub>1</sub>
V <sub>5</sub>	1	Vrai	V <sub>2</sub>
<b>V<sub>6</sub></b>	<b>3</b>	<b>Vrai</b>	<b>V<sub>3</sub></b>

# Arbre équivalent

(parcours par niveaux)

---



# Algorithme amélioré (?)

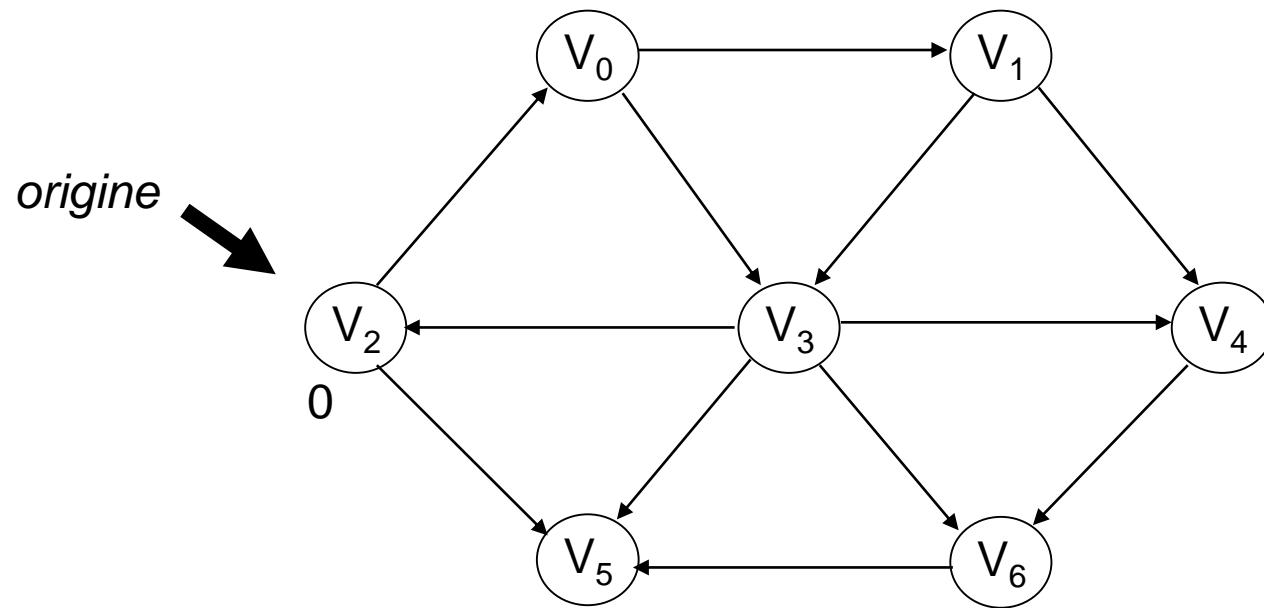
---

```
void unweighted( Vertex s ) {  
    Queue<Vertex> q = new Queue<Vertex>();  
    for each Vertex v  
        v.dist = INFINITY;  
    s.dist = 0;  
    q.enqueue( s );  
    while( !q.isEmpty( ) ) {  
        Vertex v = q.dequeue();  
        for each Vertex w adjacent to v  
            if( w.dist == INFINITY ) {  
                w.dist = v.dist + 1;  
                w.path = v;  
                q.enqueue( w );  
            }  
    }  
}
```

- Complexité:  $O(|E| + |V|)$

# Exemple avec file

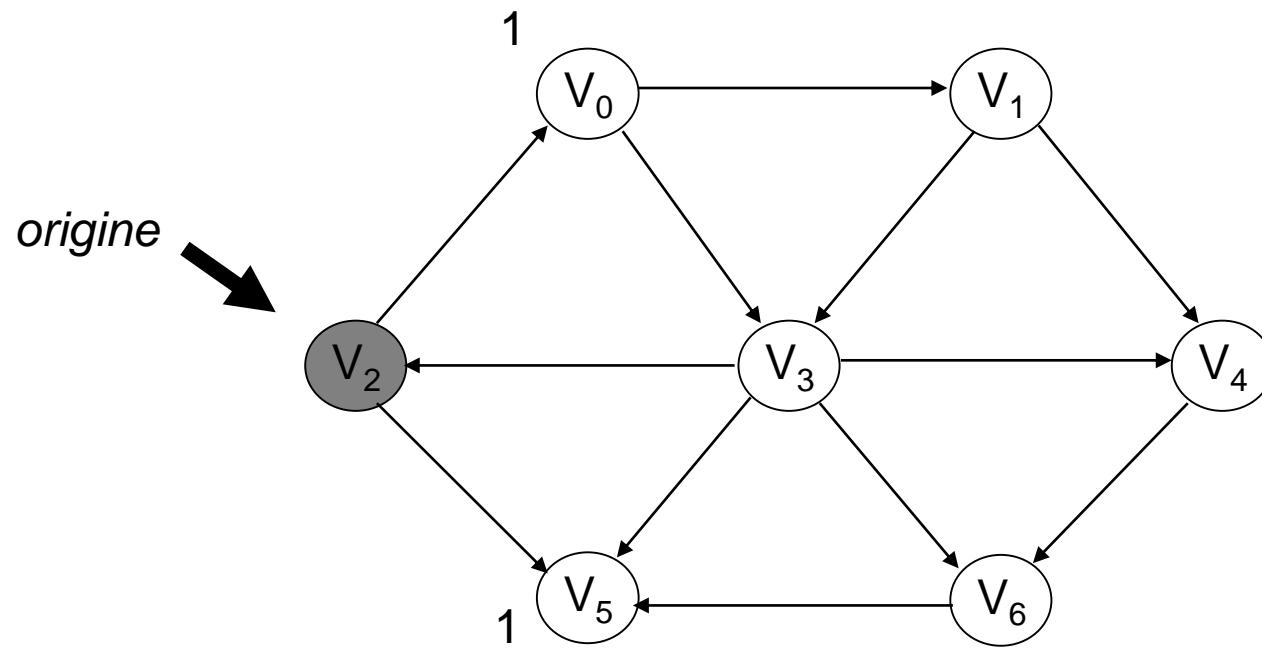
---



File:  $V_2$

# Exemple avec file

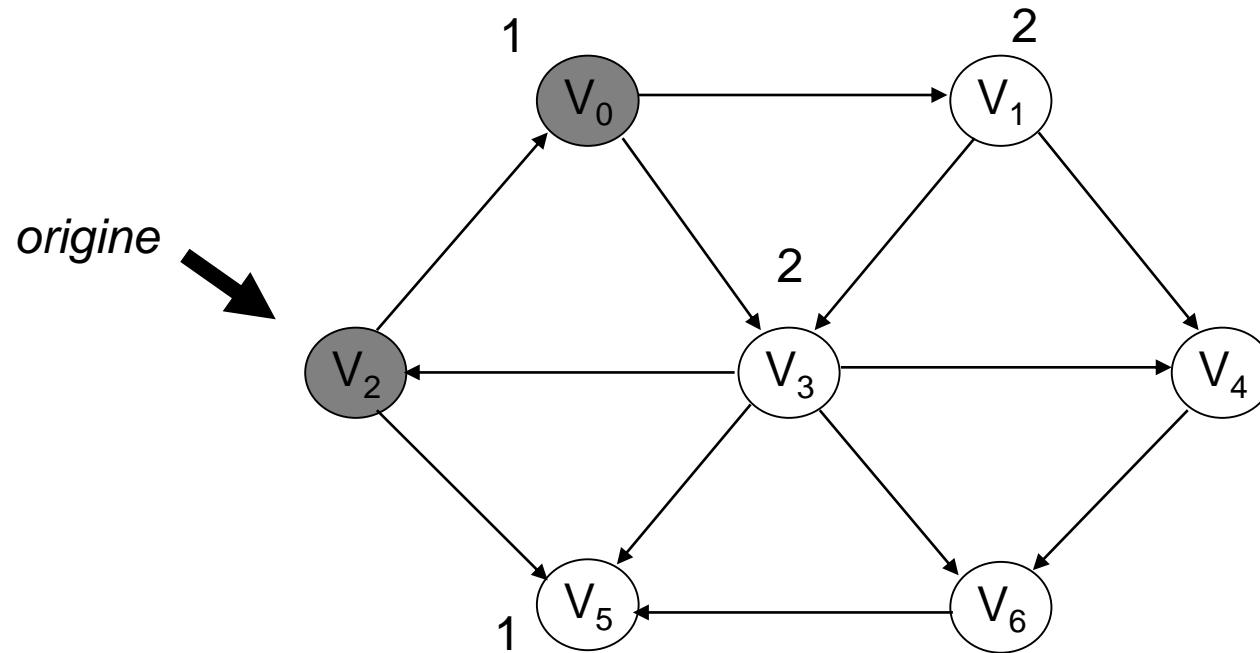
---



File:  $V_0 \ V_5$

# Exemple avec file

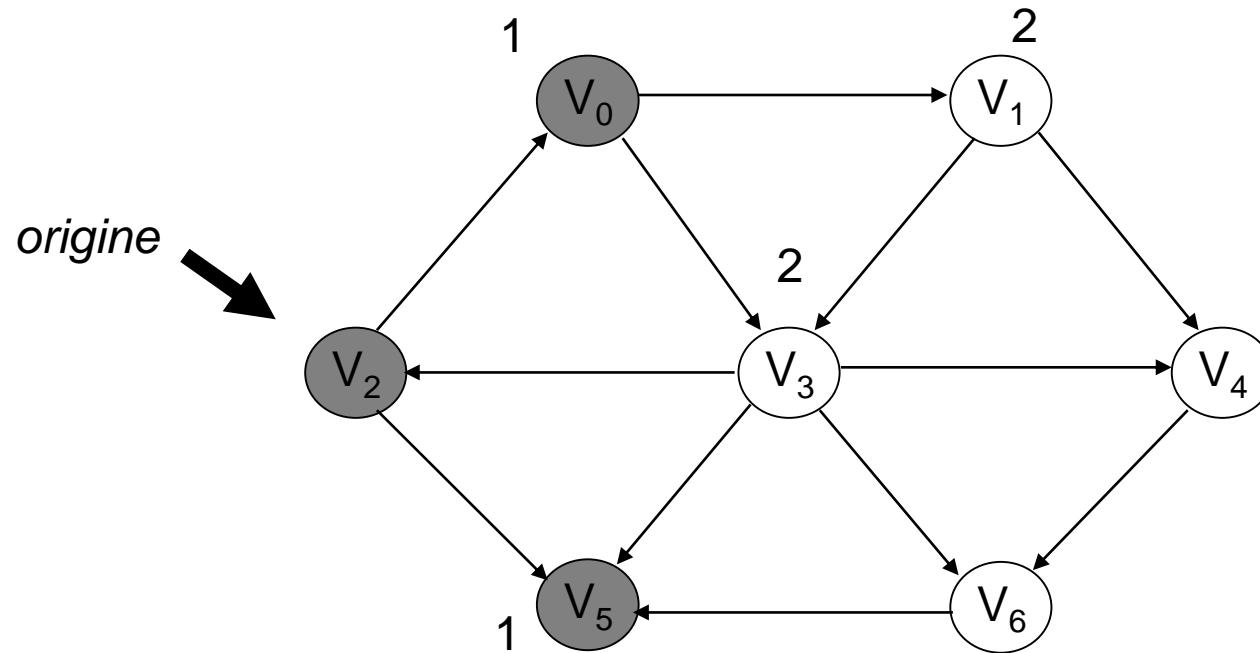
---



File:  $V_5 \ V_1 \ V_3$

# Exemple avec file

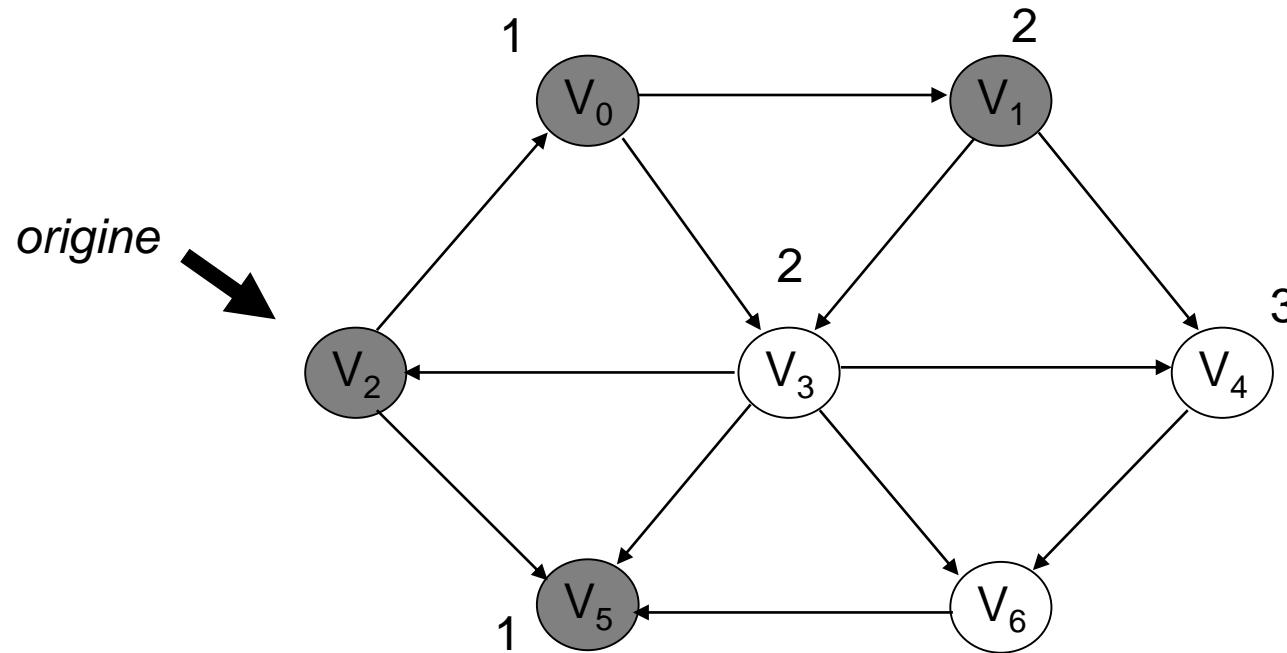
---



File:  $V_1$   $V_3$

# Exemple avec file

---

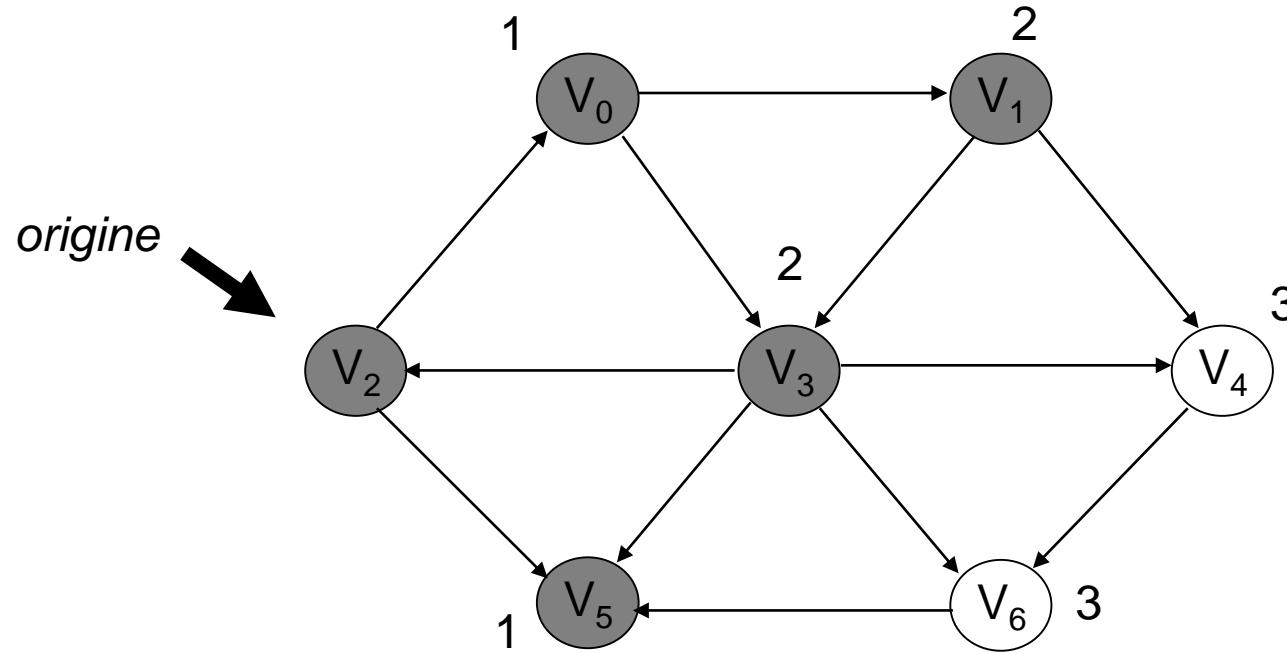


File:  $V_3$   $V_4$

---

# Exemple avec file

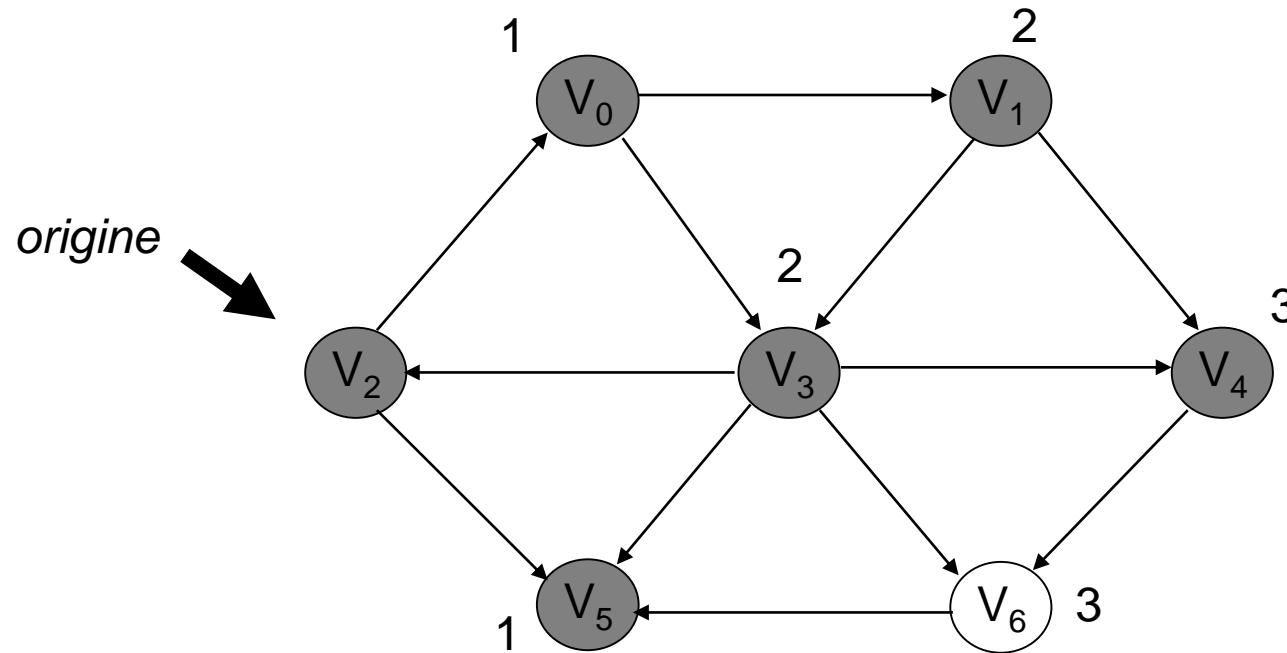
---



File:  $v_4 \ v_6$

# Exemple avec file

---

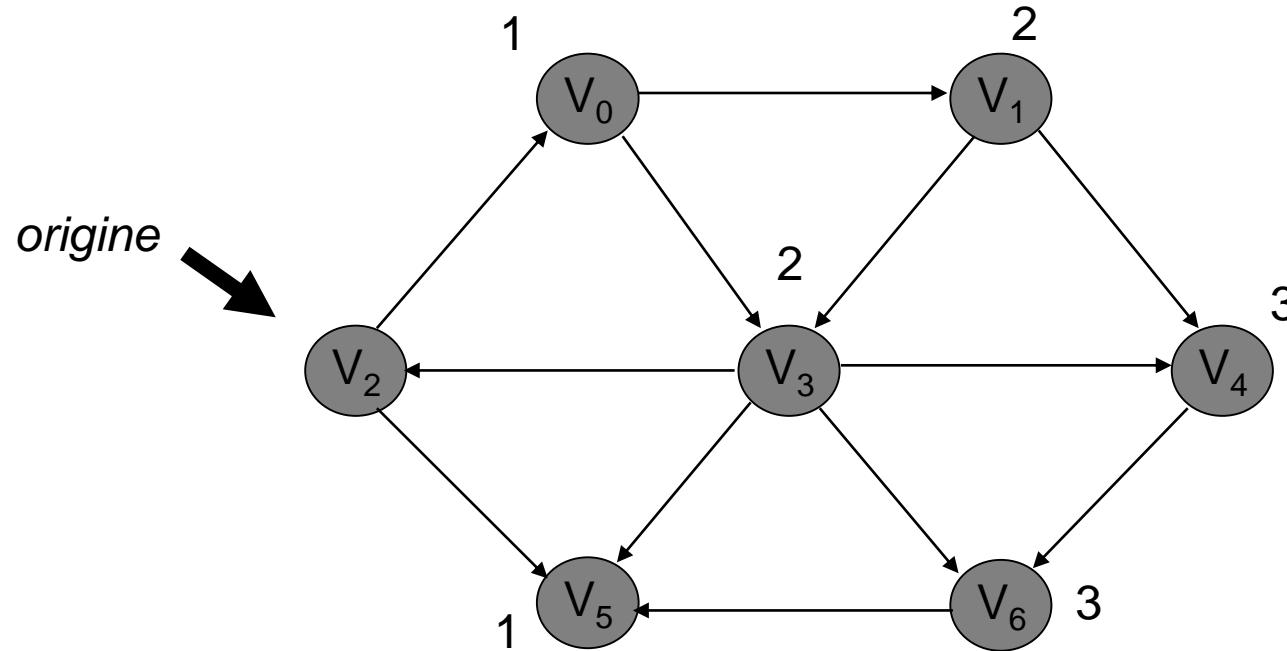


File:  $V_6$

---

# Exemple avec file

---



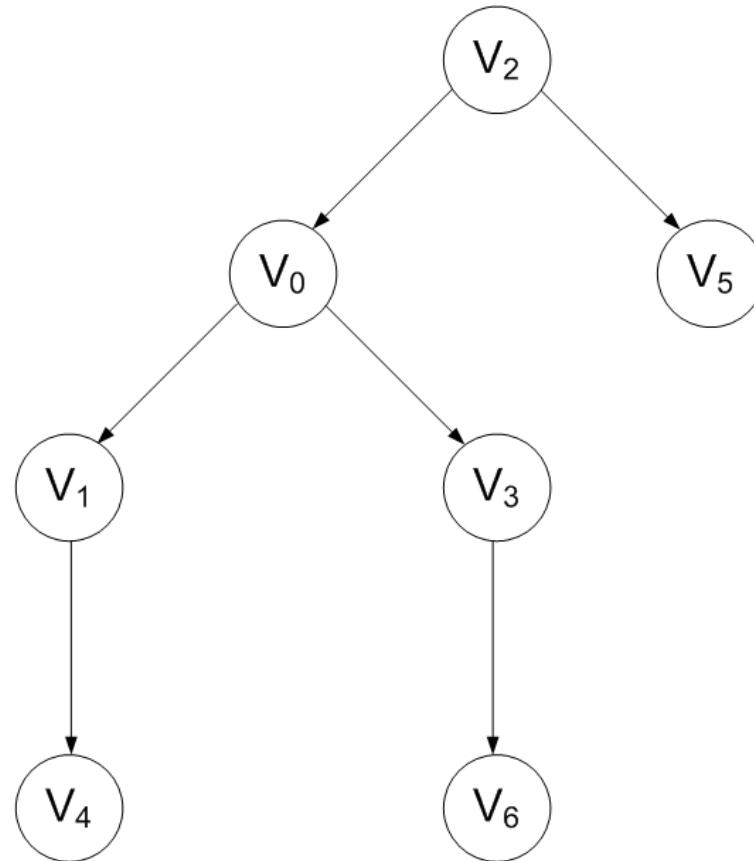
File: vide

---

# Arbre équivalent

(parcours par niveaux)

---



# Graphes

---

- 1.Définitions et exemples
- 2.Implémentations
- 3.Ordre topologique
- 4.Chemin le plus court
- 5.Dijkstra**
- 6.Parcours

# Plus court chemin avec poids

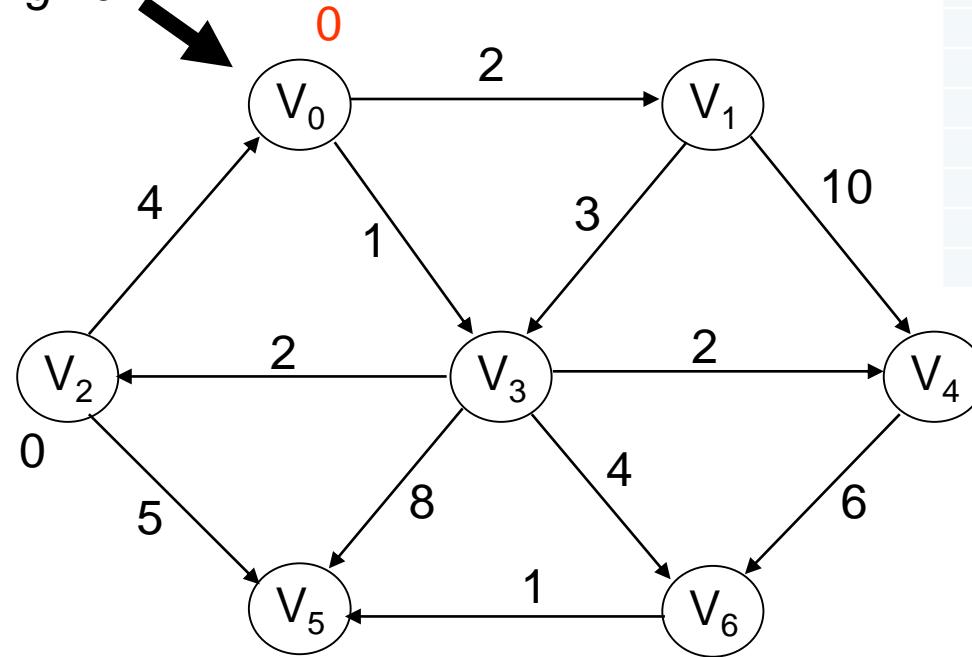
---

- Graphe orienté
- Nœud de départ
- Coût associé aux arêtes
  - Poids (non négatif)

# Algorithme de Dijkstra

(avec file de priorité)

origine



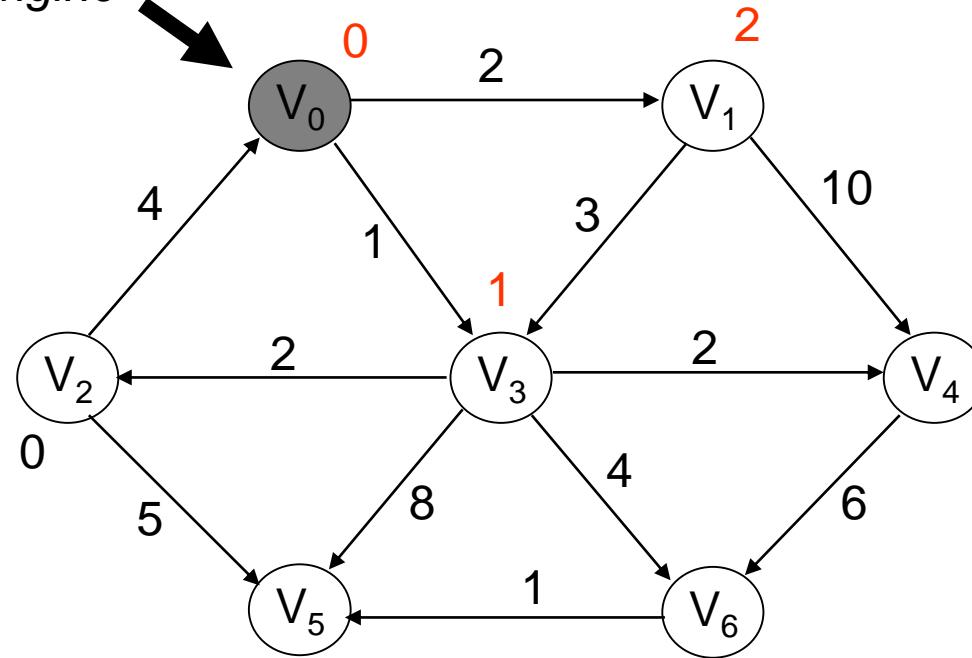
	Cout	Parent	Connu?
v0	0	-	X
v1	2	v0	X
v2	3	v3	X
v3	1	v0	X
v4	3	v3	X
v5	6	v6	X
v6	5	v3	X

File de priorité:  $(V_0, 0)$

# Algorithme de Dijkstra

(avec file de priorité)

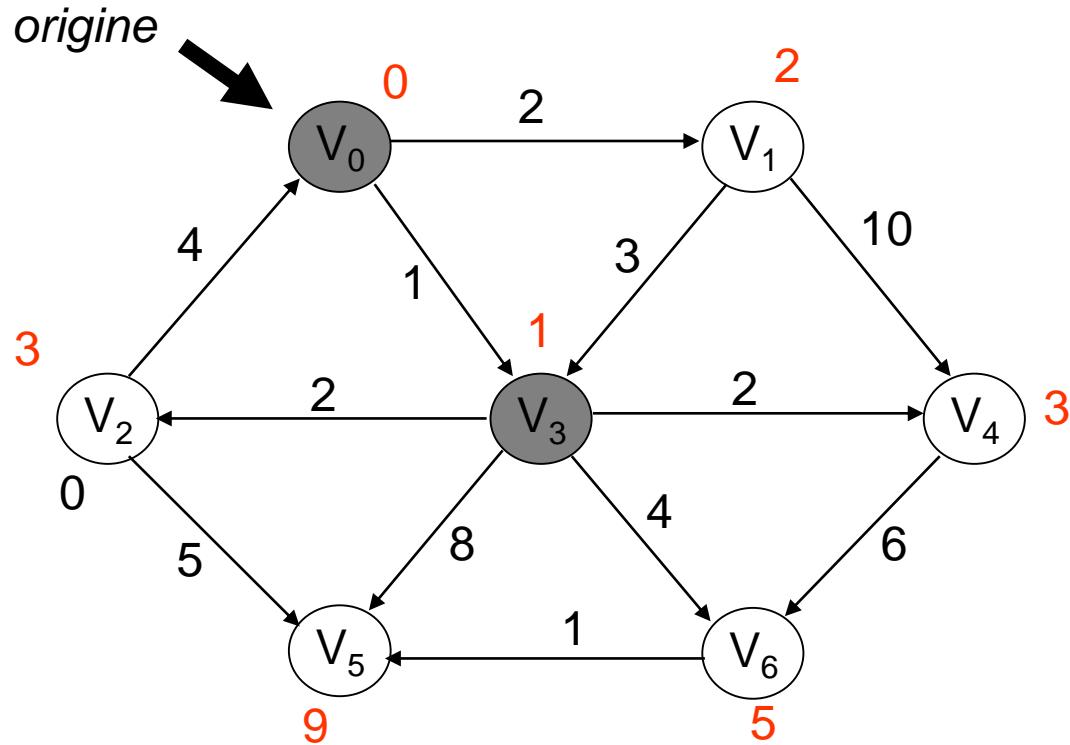
*origine*



File de priorité:  $(V_3, 1)$   $(V_1, 2)$

# Algorithme de Dijkstra

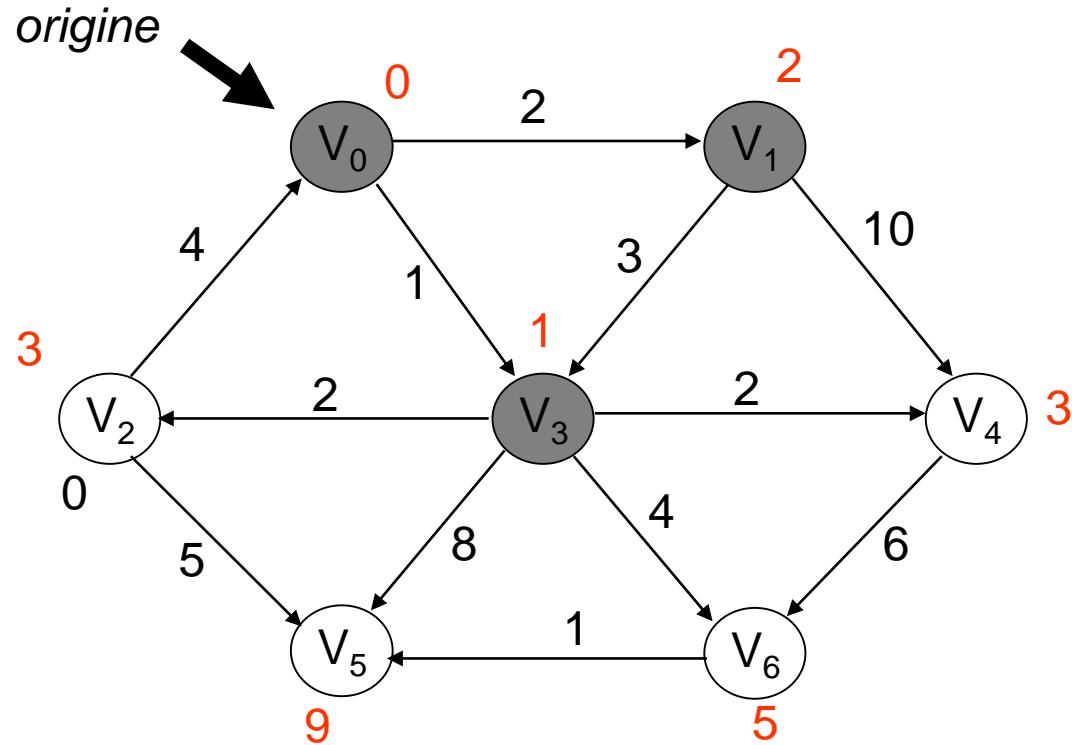
(avec file de priorité)



File de priorité:  $(V_1, 2) (V_2, 3) (V_4, 3) (V_6, 5) (V_5, 9)$

# Algorithme de Dijkstra

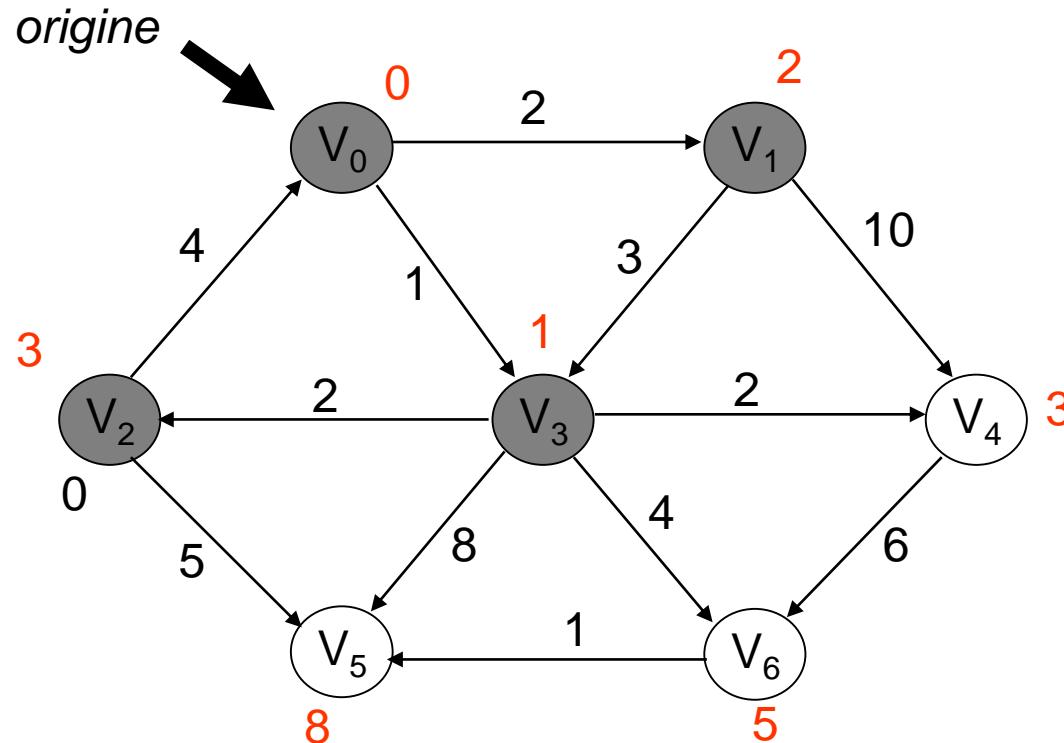
(avec file de priorité)



File de priorité:  $(V_2, 3)$   $(V_4, 3)$   $(V_6, 5)$   $(V_5, 9)$

# Algorithme de Dijkstra

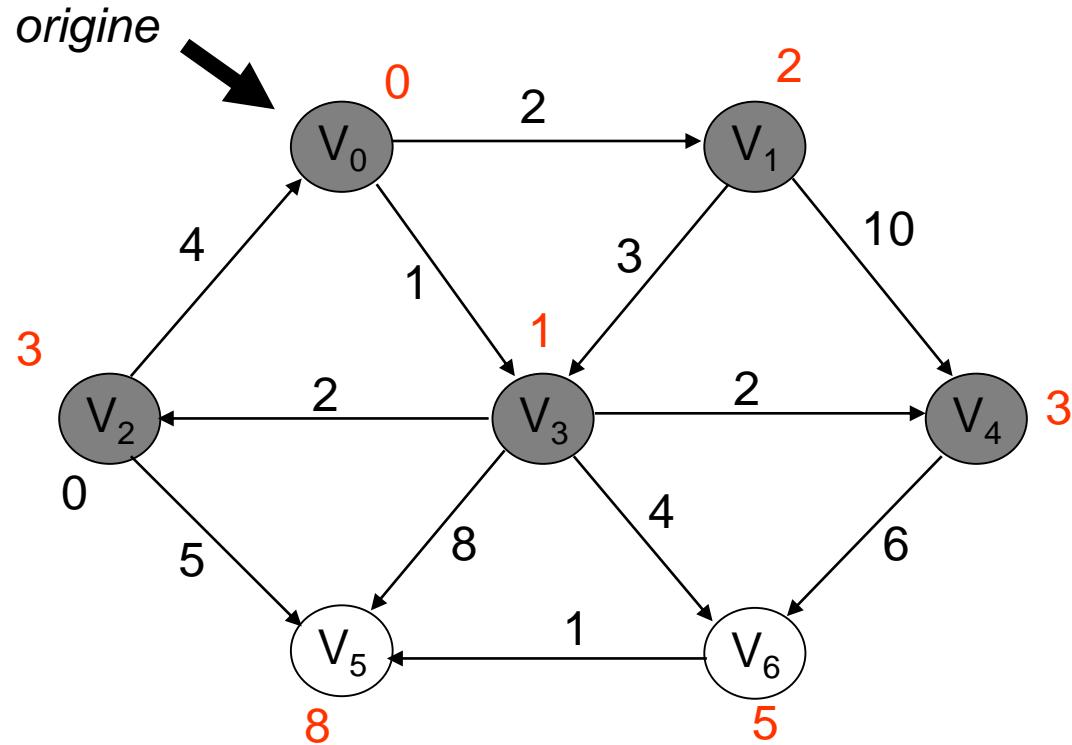
(avec file de priorité)



File de priorité:  $(V_4, 3) (V_6, 5) (V_5, 8)$

# Algorithme de Dijkstra

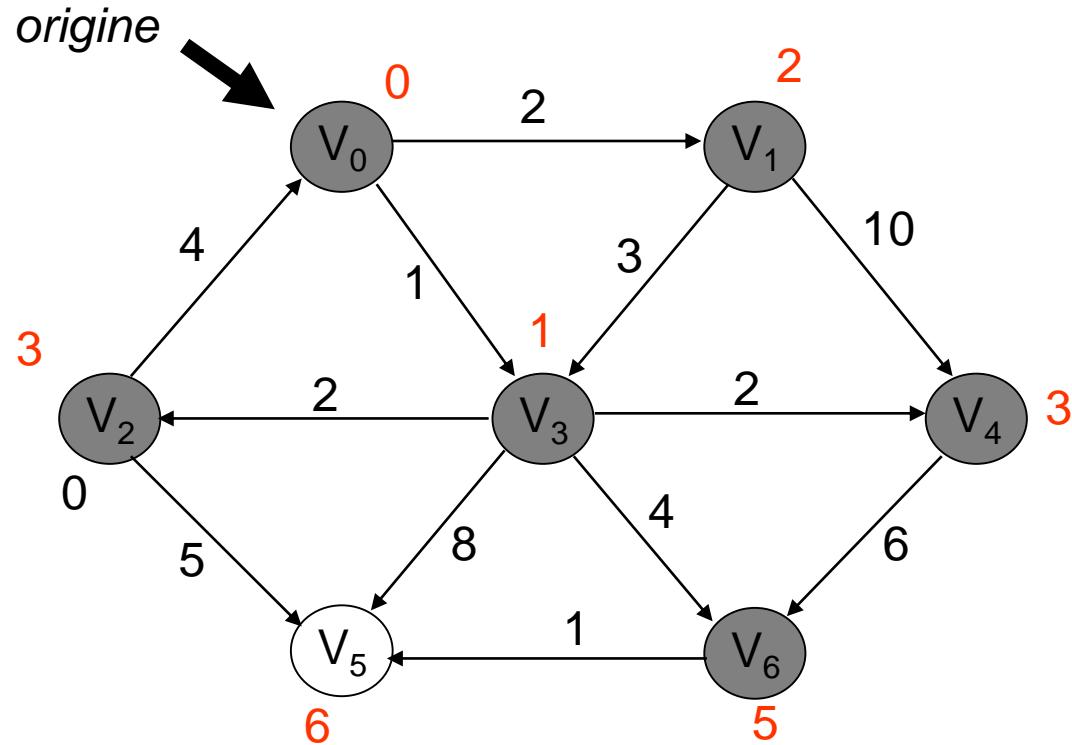
(avec file de priorité)



File de priorité:  $(V_6, 5)$   $(V_5, 8)$

# Algorithme de Dijkstra

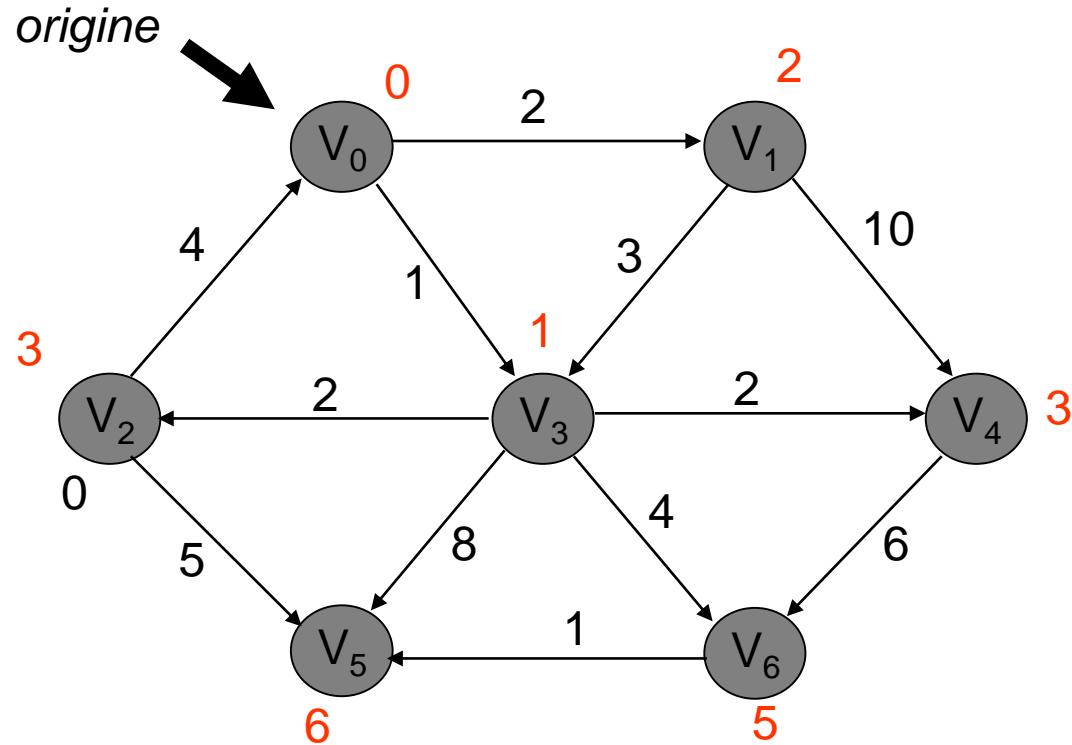
(avec file de priorité)



File de priorité:  $(V_5, 8)$

# Algorithme de Dijkstra

(avec file de priorité)



File de priorité: Vide

# Graphes

---

- 1.Définitions et exemples
- 2.Implémentations
- 3.Ordre topologique
- 4.Chemin le plus court
- 5.Dijkstra
- 6.Parcours

# Algorithmes de visite

---

1. Breadth First Search (équivalent à par niveau)

Vu au tri topologique

2. Depth First Search (équivalent à pré-ordre)

On part d'un nœud,

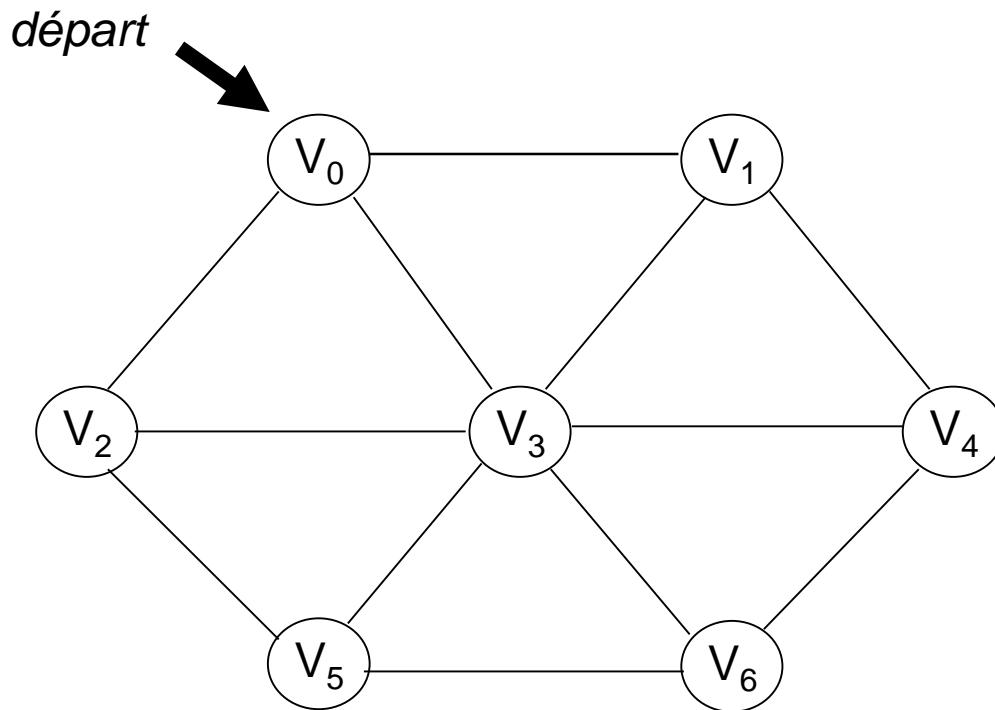
Visite ses enfants

Pour chacun de ses enfants, on refait la même chose

Chaque nœud visité est marqué comme tel

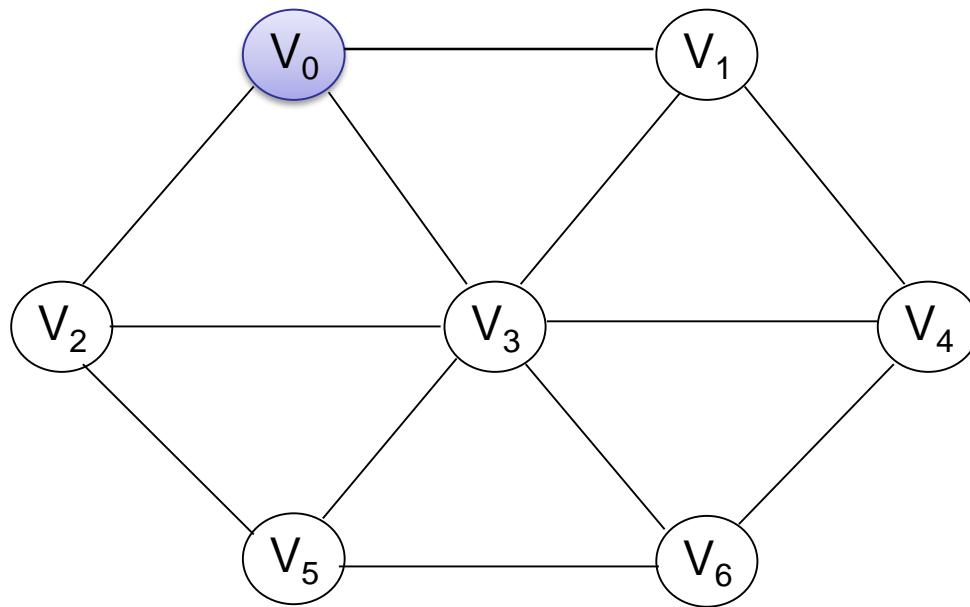
# Ex. 1 BFS – graphe non orienté

---



# Ex. 1 BFS – graphe non orienté

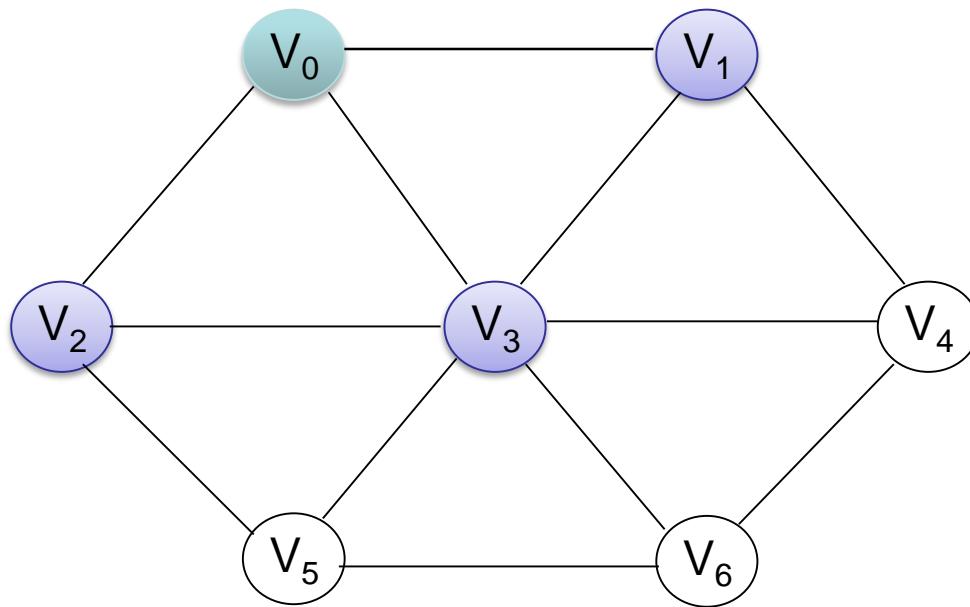
---



$V_0,$

# Ex. 1 BFS – graphe non orienté

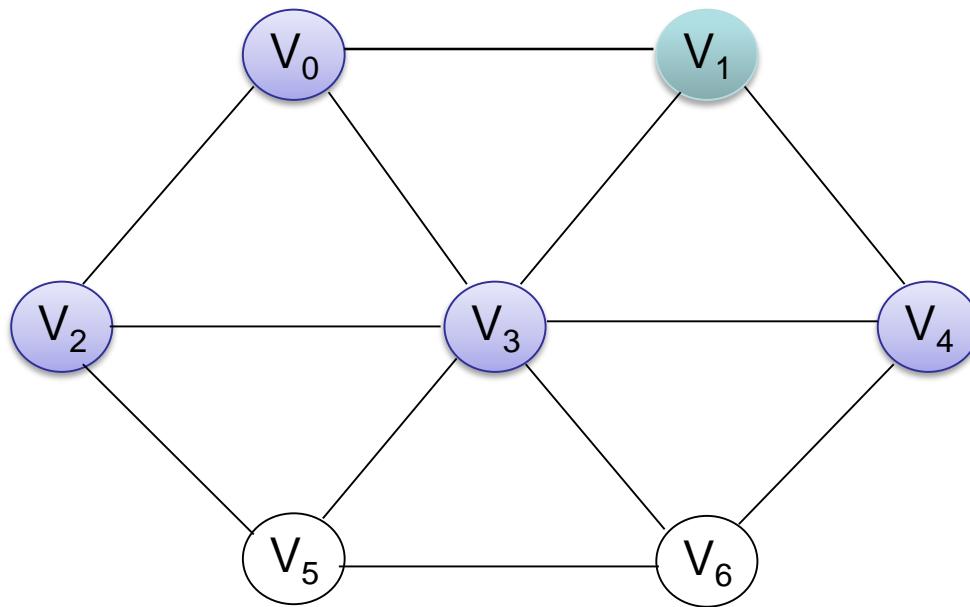
---



$V_0, V_1, V_2, V_3,$

# Ex. 1 BFS – graphe non orienté

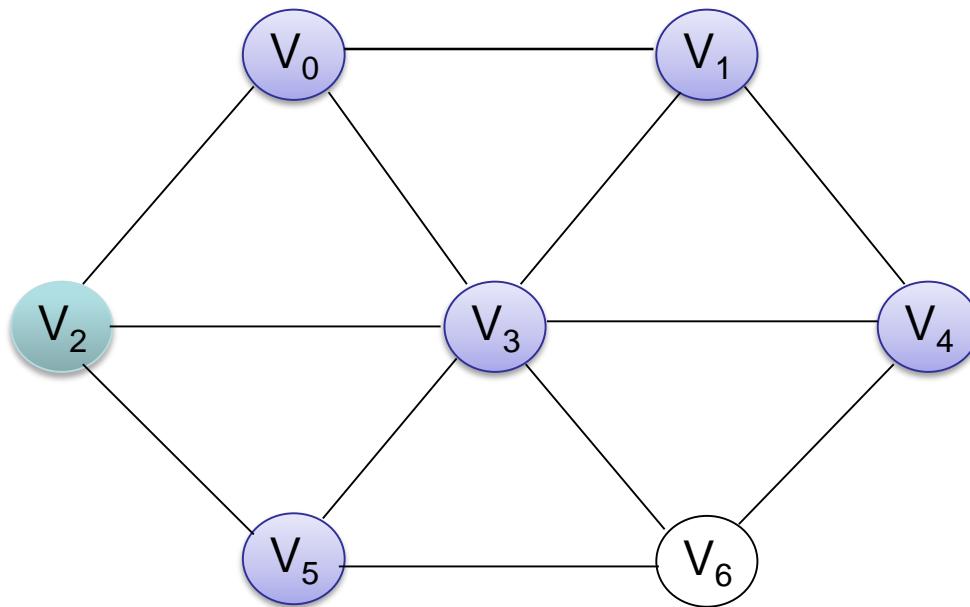
---



$V_0, V_1, V_2, V_3, V_4,$

# Ex. 1 BFS – graphe non orienté

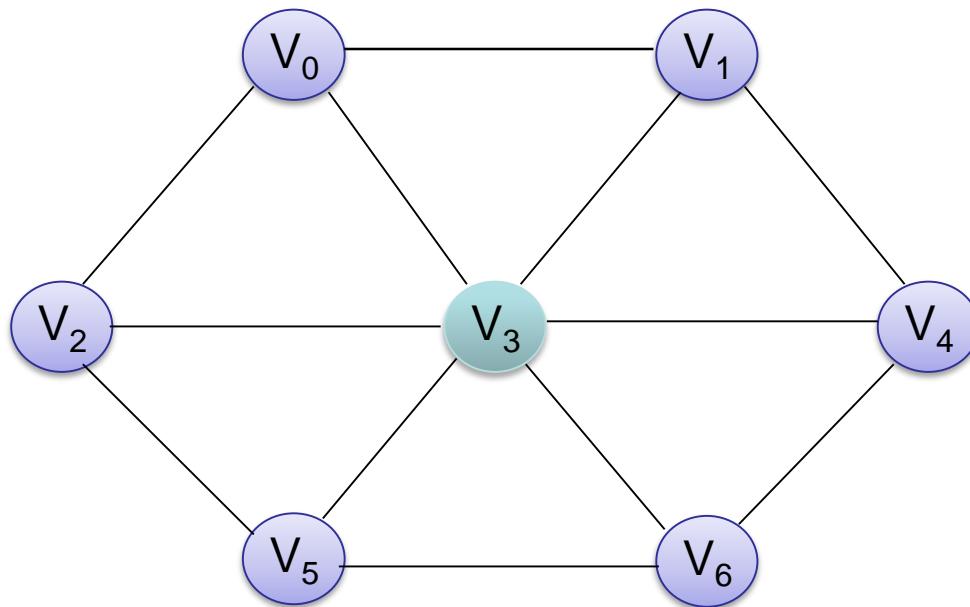
---



$V_0, V_1, V_2, V_3, V_4, V_5,$

# Ex. 1 BFS – graphe non orienté

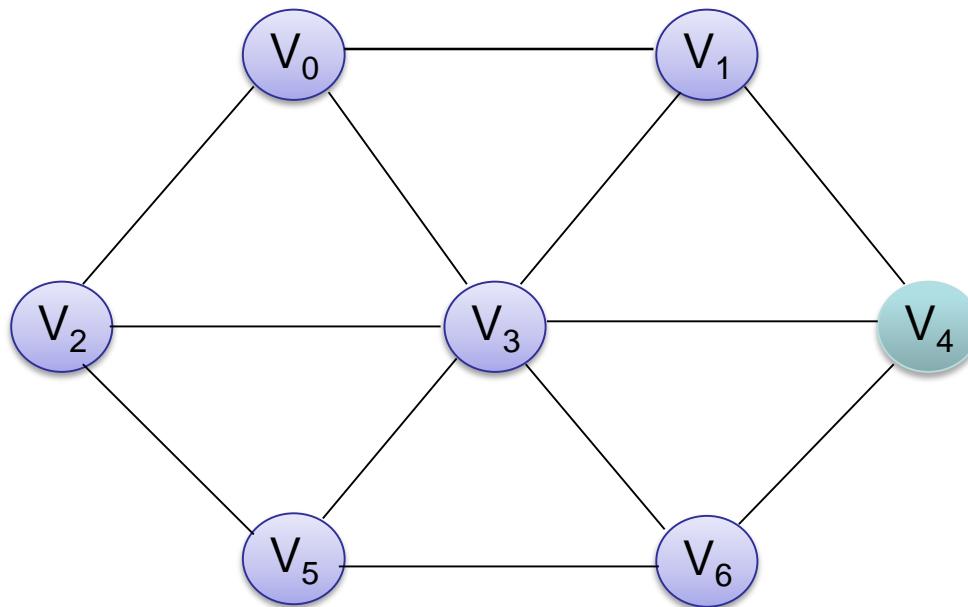
---



$V_0, V_1, V_2, V_3, V_4, V_5, V_6$

# Ex. 1 BFS – graphe non orienté

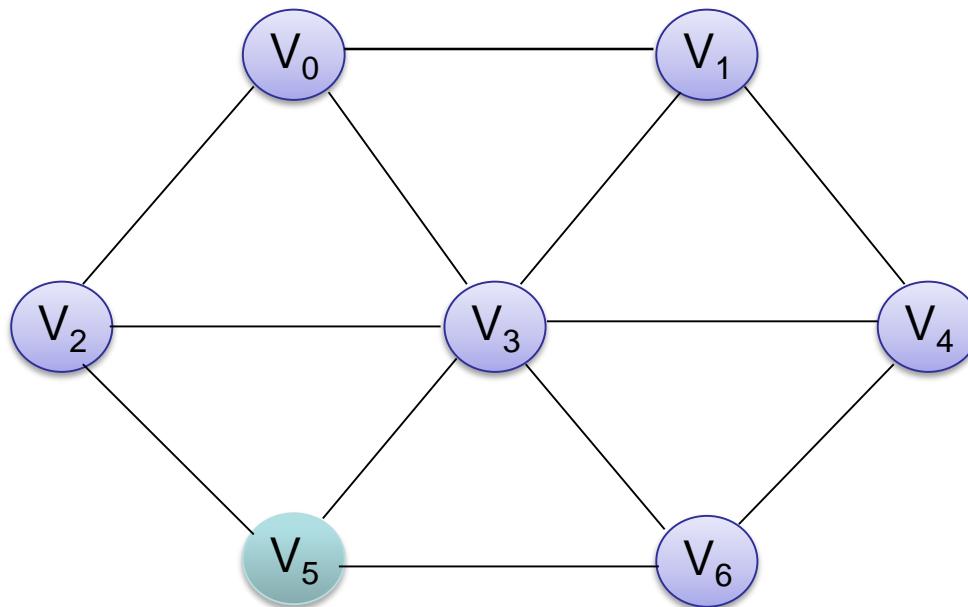
---



$V_0, V_1, V_2, V_3, V_4, V_5, V_6$

# Ex. 1 BFS – graphe non orienté

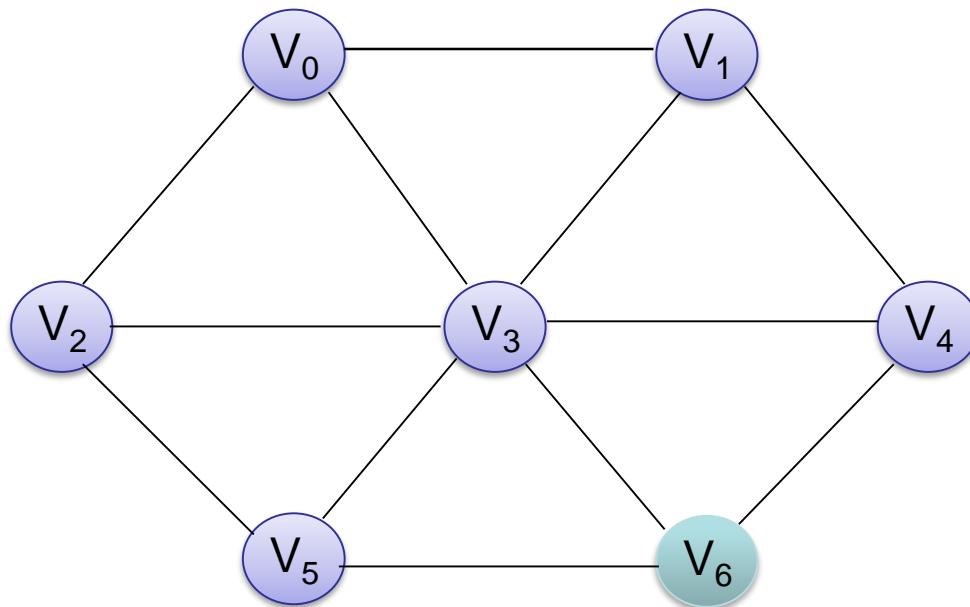
---



$V_0, V_1, V_2, V_3, V_4, V_5, V_6$

# Ex. 1 BFS – graphe non orienté

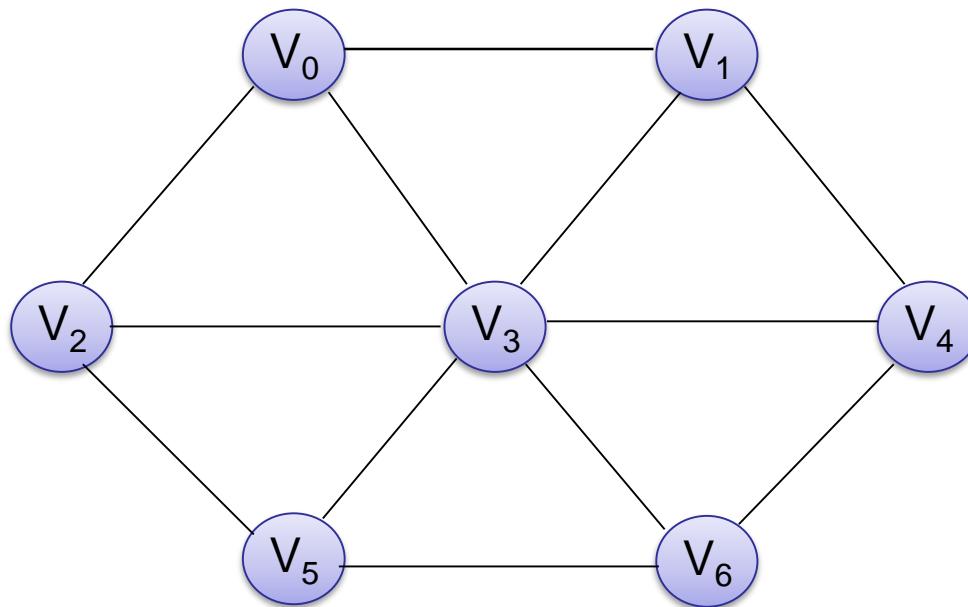
---



$V_0, V_1, V_2, V_3, V_4, V_5, V_6$

# Ex. 1 BFS – graphe non orienté

---

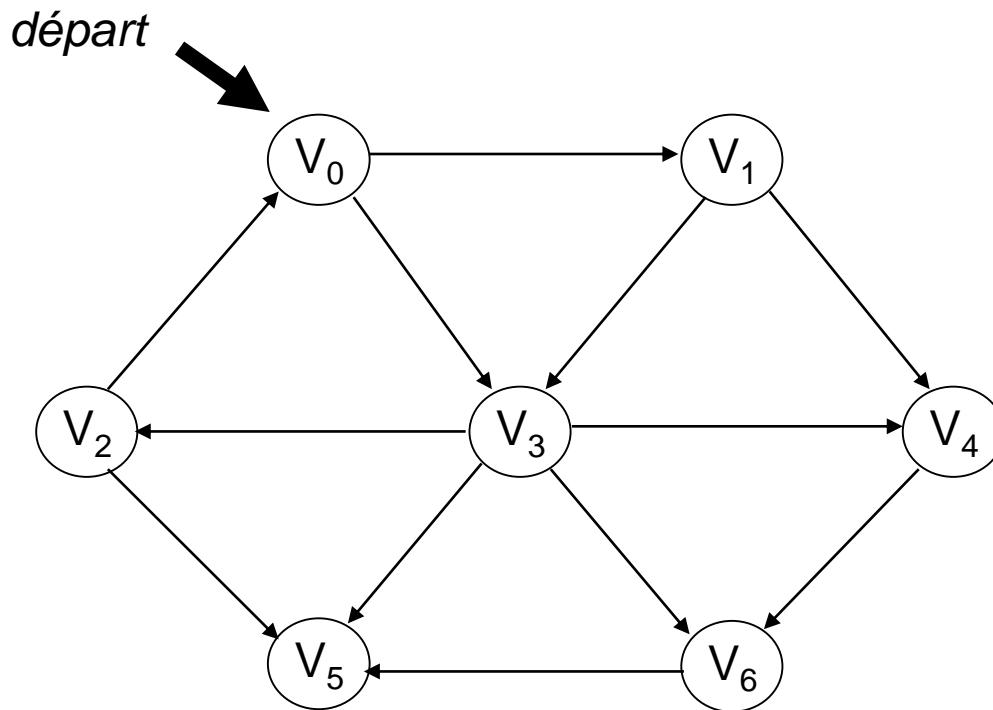


$V_0, V_1, V_2, V_3, V_4, V_5, V_6$ . FIN

---

# Ex. 2 BFS – graphe orienté

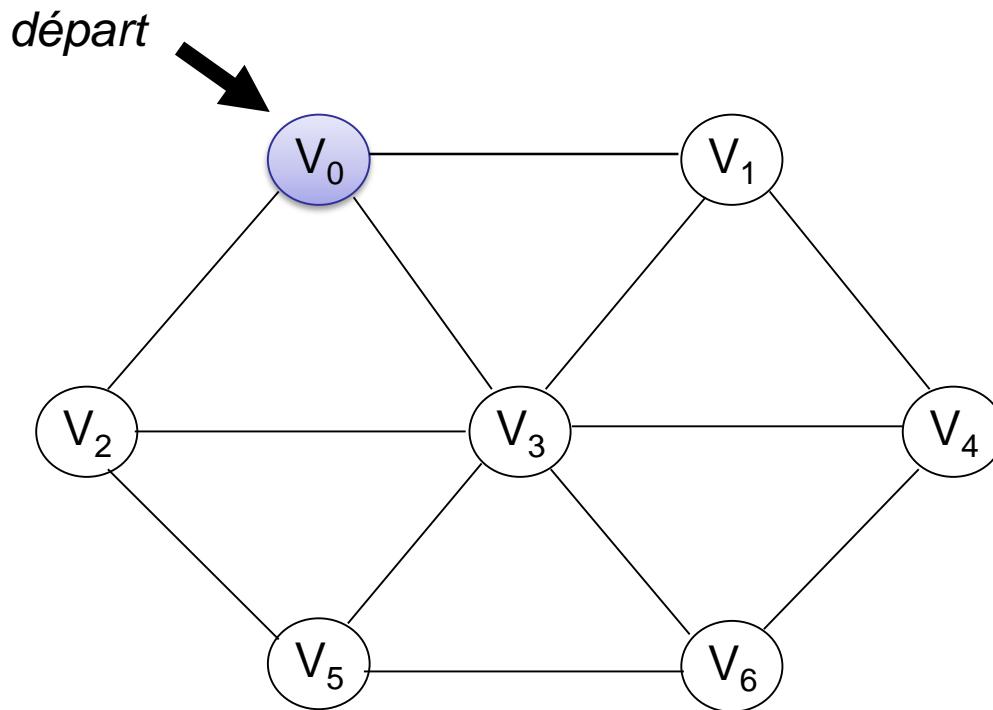
---



$V_0, V_1, V_3, V_4, V_2, V_5, V_6.$

# Ex. 3 DFS – graphe non orienté

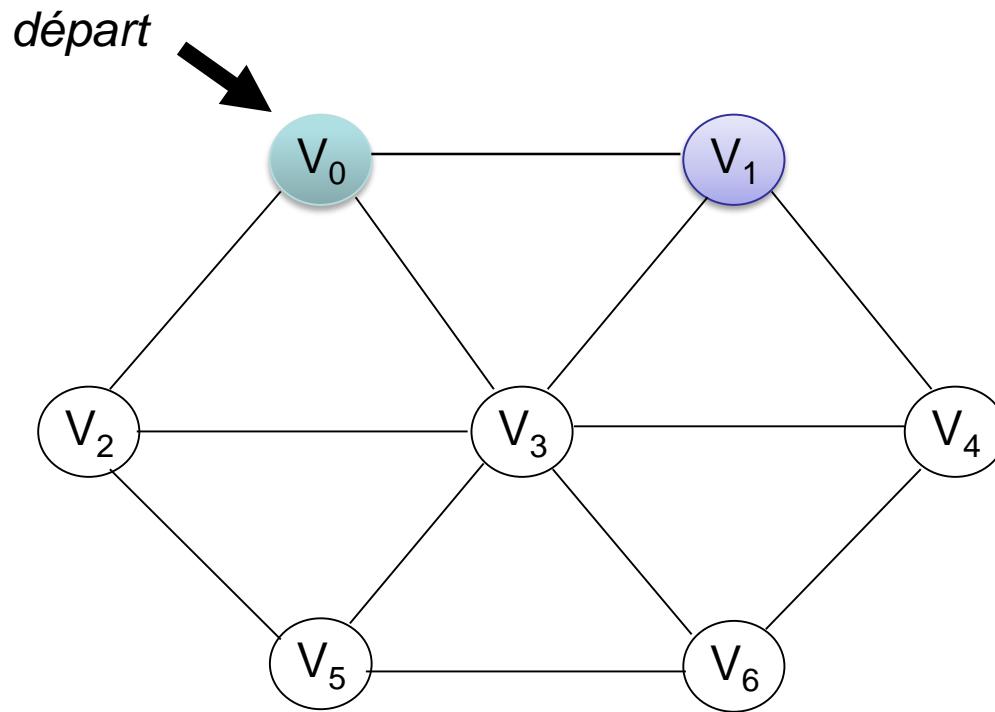
---



$V_0,$

# Ex. 3 DFS – graphe non orienté

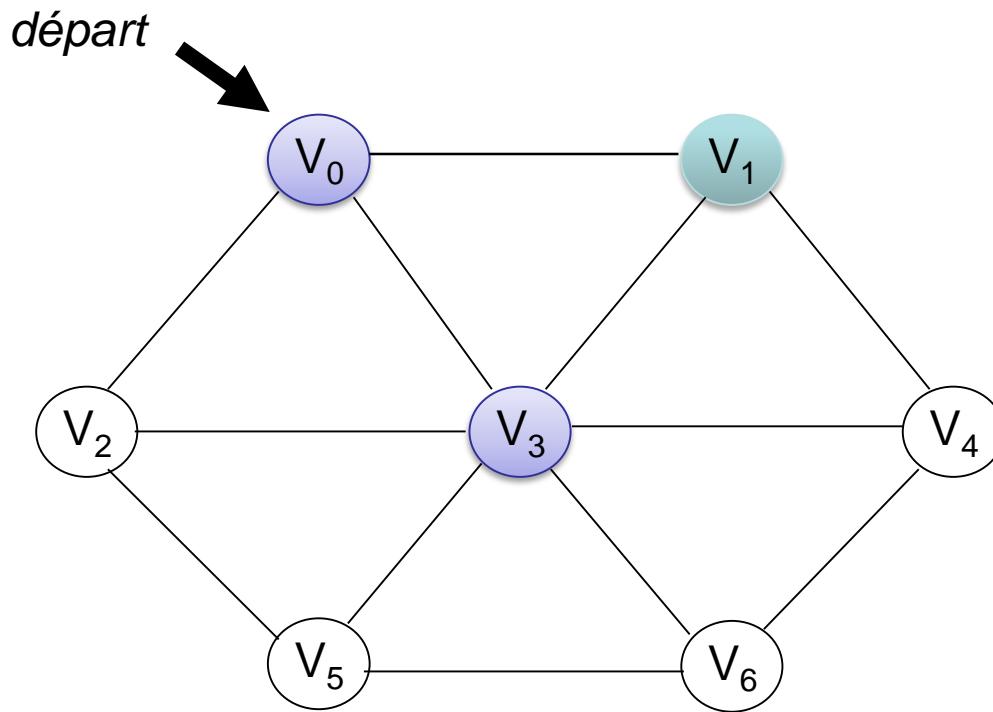
---



$V_0, V_1,$

# Ex. 3 DFS – graphe non orienté

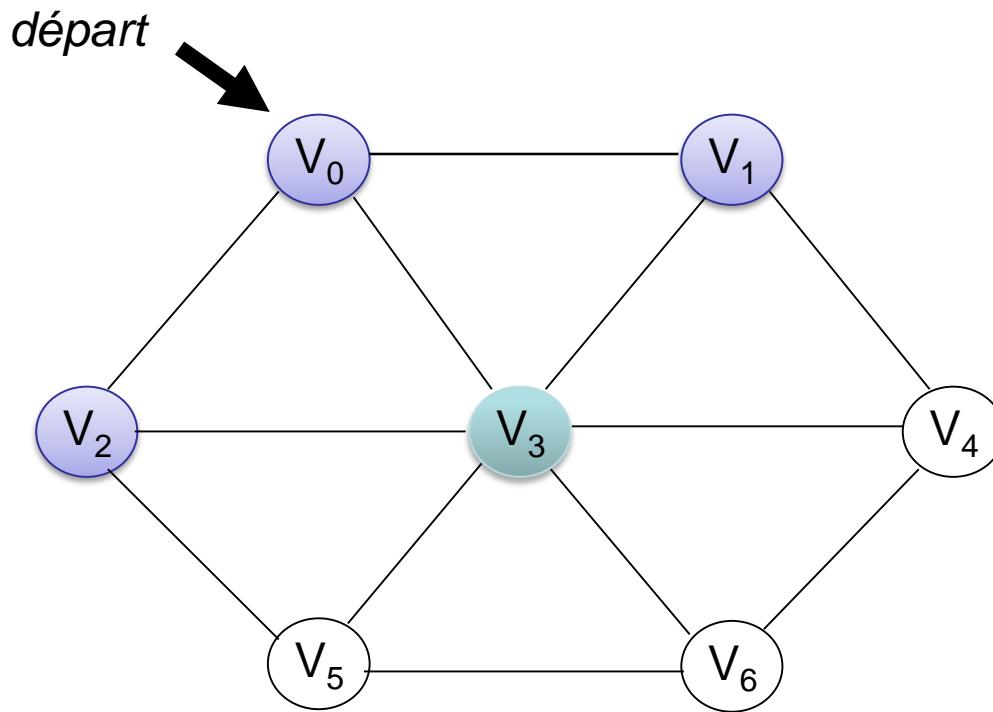
---



$V_0, V_1, V_3,$

# Ex. 3 DFS – graphe non orienté

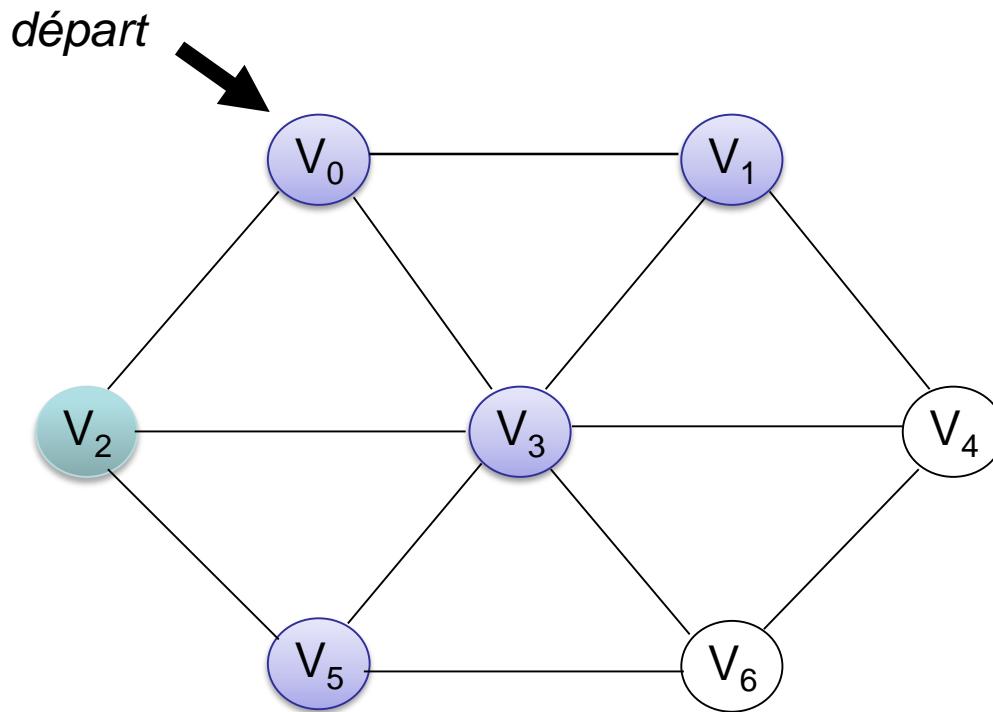
---



$V_0, V_1, V_3, V_2,$

# Ex. 3 DFS – graphe non orienté

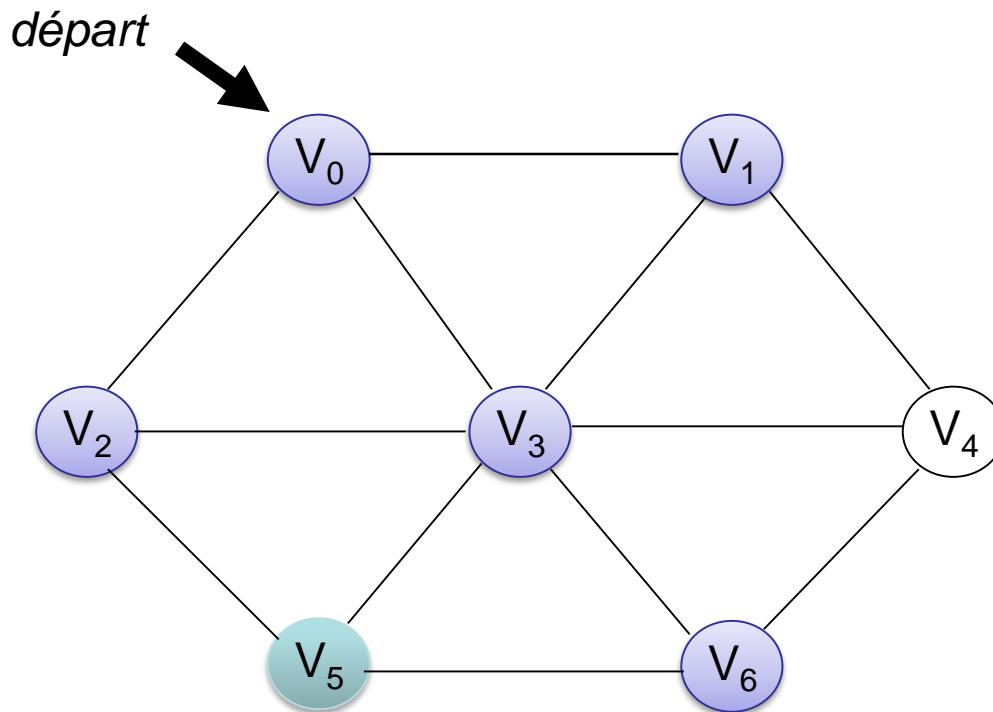
---



$V_0, V_1, V_3, V_2, V_5,$

# Ex. 3 DFS – graphe non orienté

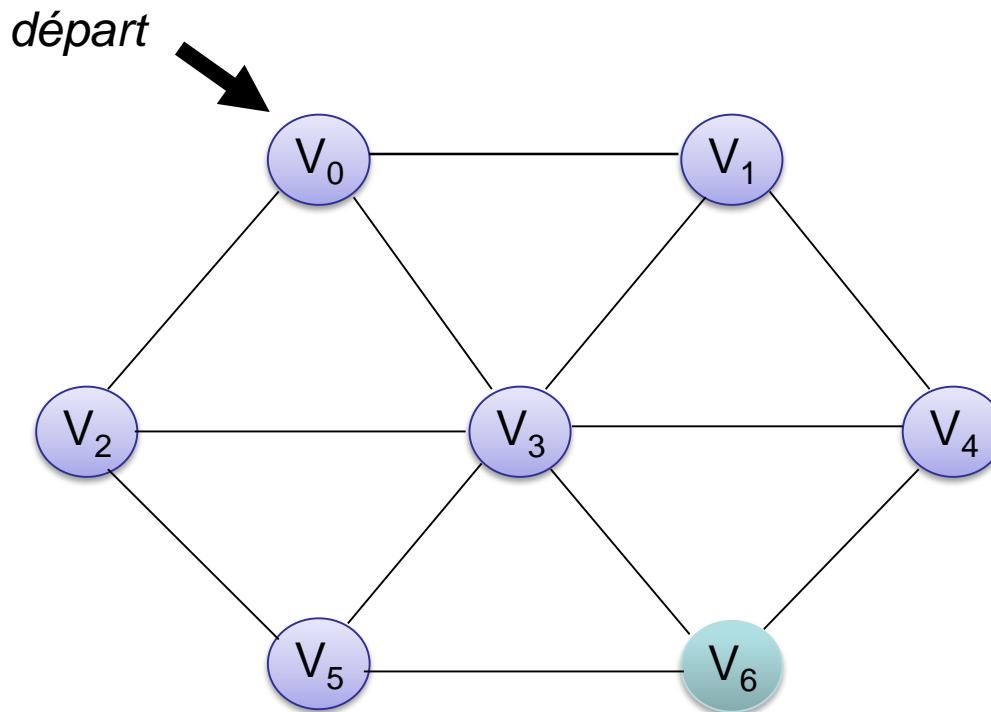
---



$V_0, V_1, V_3, V_2, V_5, V_6,$

# Ex. 3 DFS – graphe non orienté

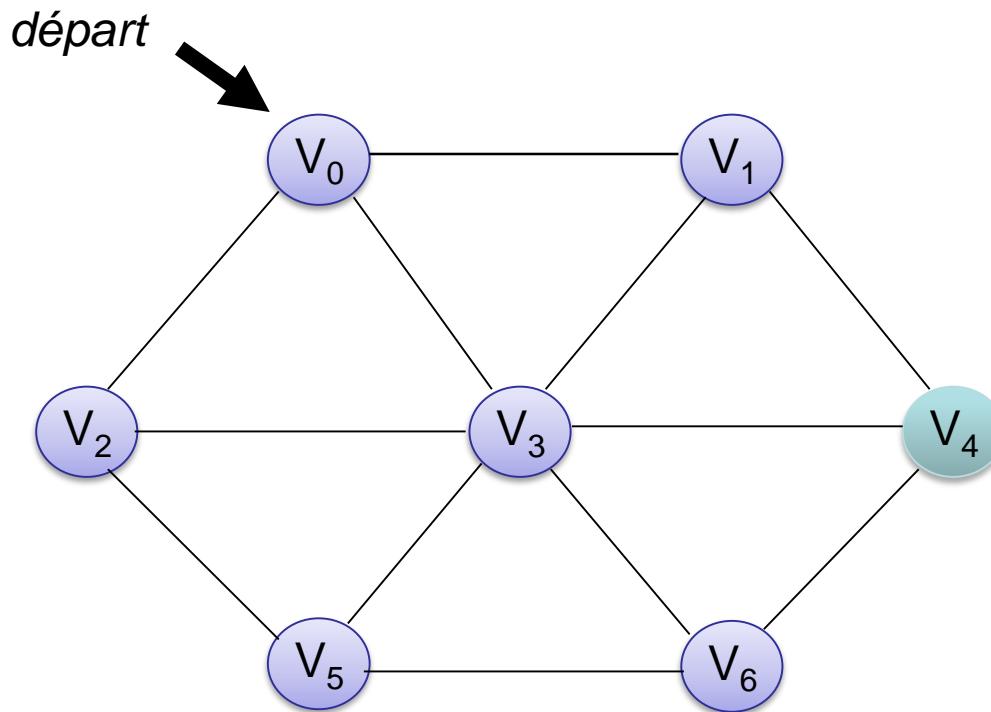
---



$V_0, V_1, V_3, V_2, V_5, V_6, V_4,$

# Ex. 3 DFS – graphe non orienté

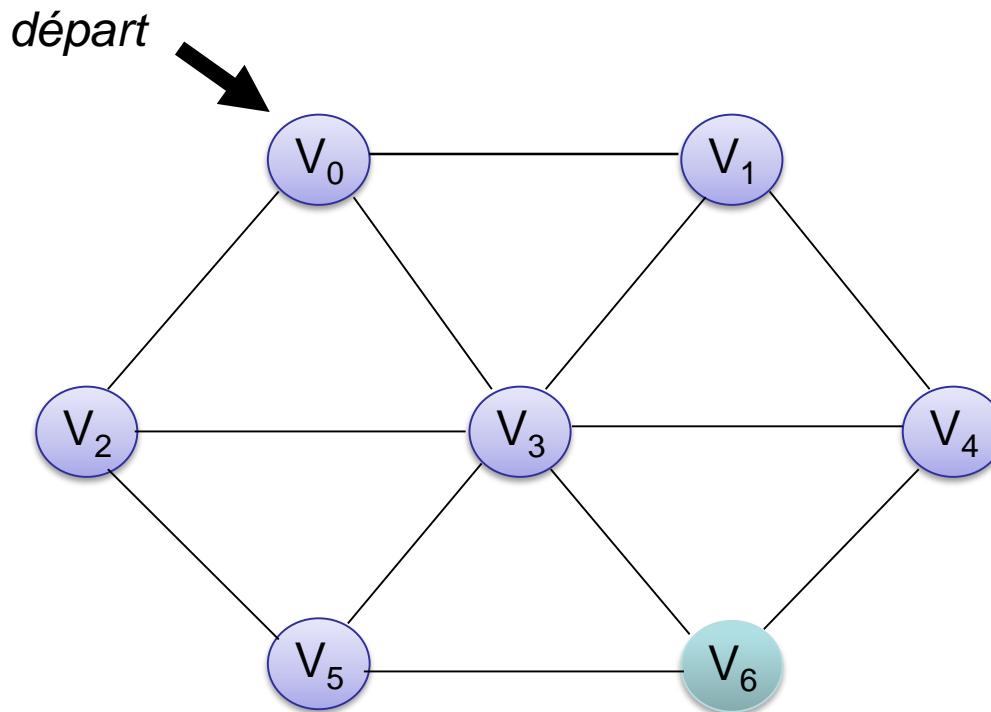
---



$V_0, V_1, V_3, V_2, V_5, V_6, V_4,$

# Ex. 3 DFS – graphe non orienté

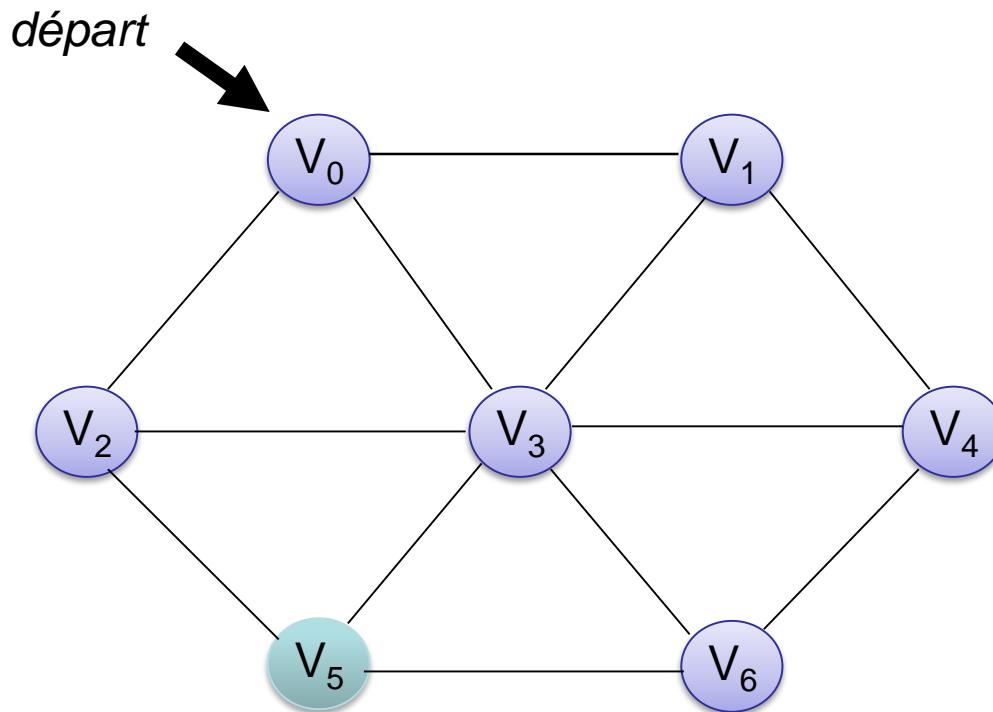
---



$V_0, V_1, V_3, V_2, V_5, V_6, V_4,$

# Ex. 3 DFS – graphe non orienté

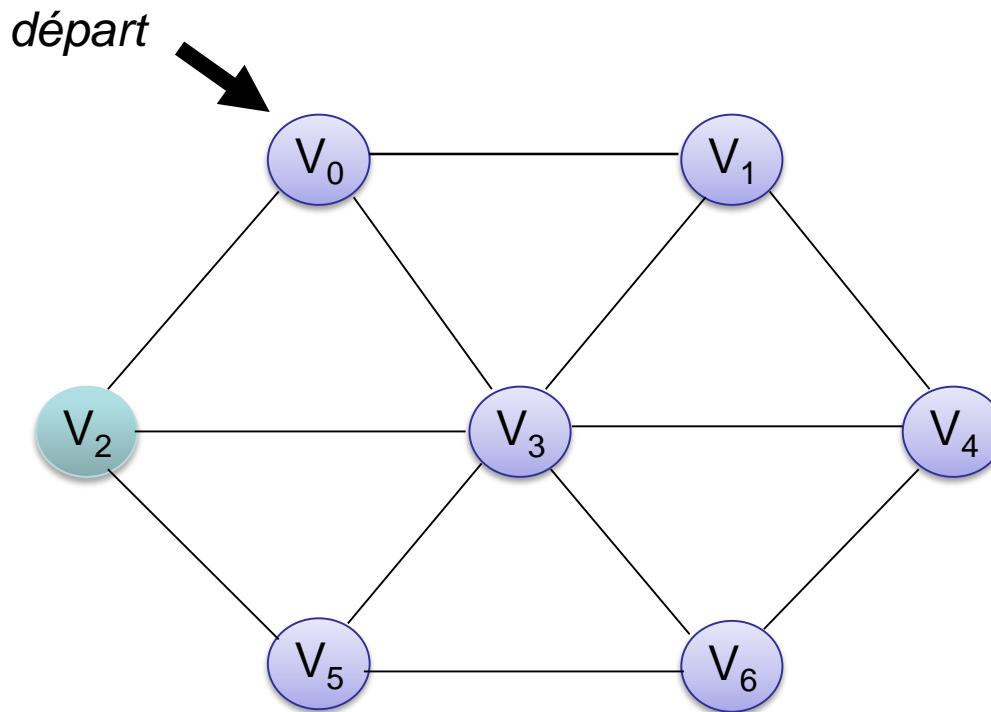
---



$V_0, V_1, V_3, V_2, V_5, V_6, V_4,$

# Ex. 3 DFS – graphe non orienté

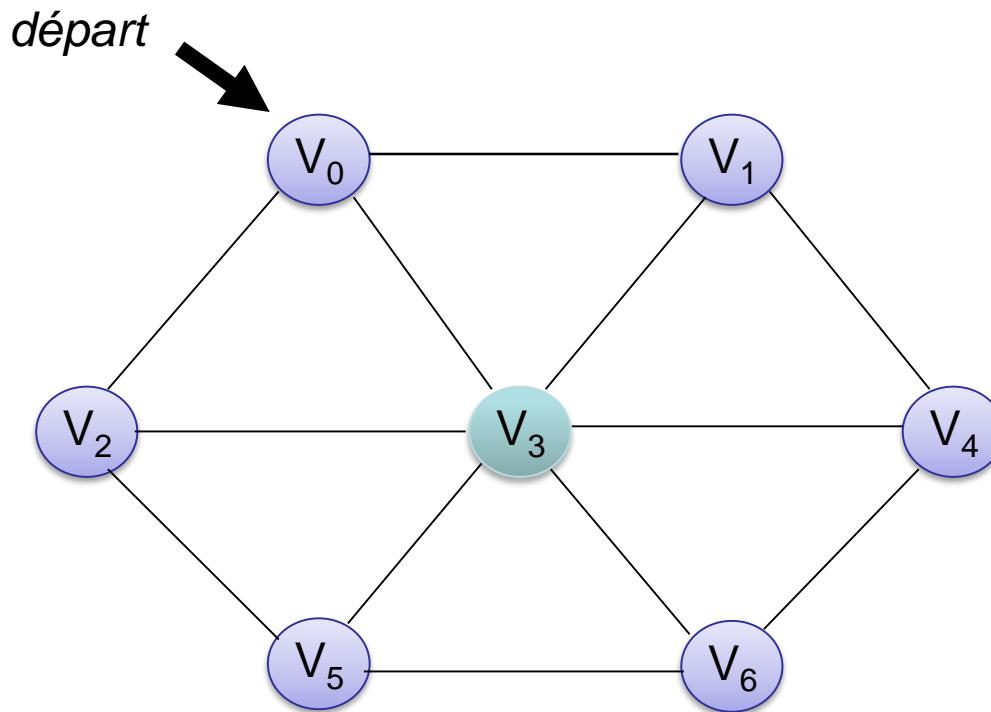
---



$V_0, V_1, V_3, V_2, V_5, V_6, V_4,$

# Ex. 3 DFS – graphe non orienté

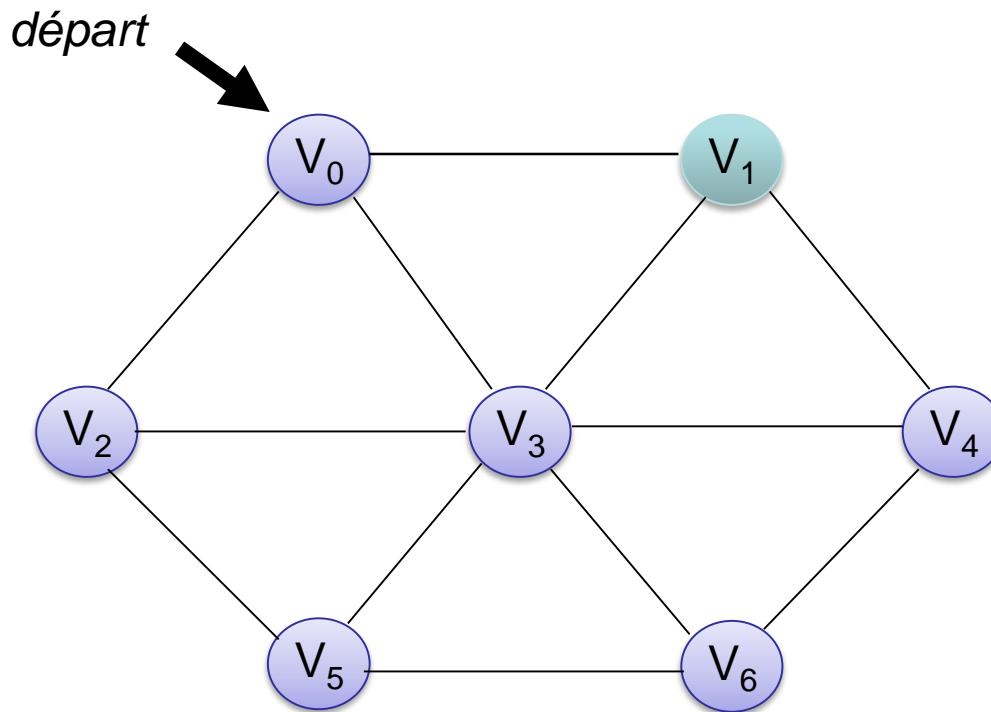
---



$V_0, V_1, V_3, V_2, V_5, V_6, V_4,$

# Ex. 3 DFS – graphe non orienté

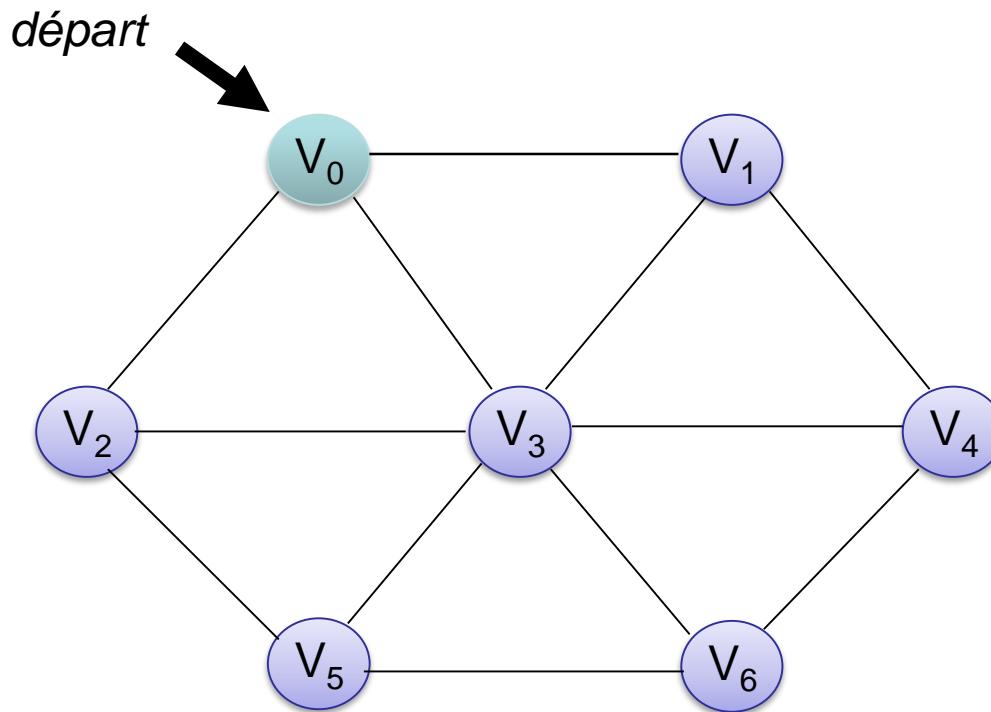
---



$V_0, V_1, V_3, V_2, V_5, V_6, V_4,$

# Ex. 3 DFS – graphe non orienté

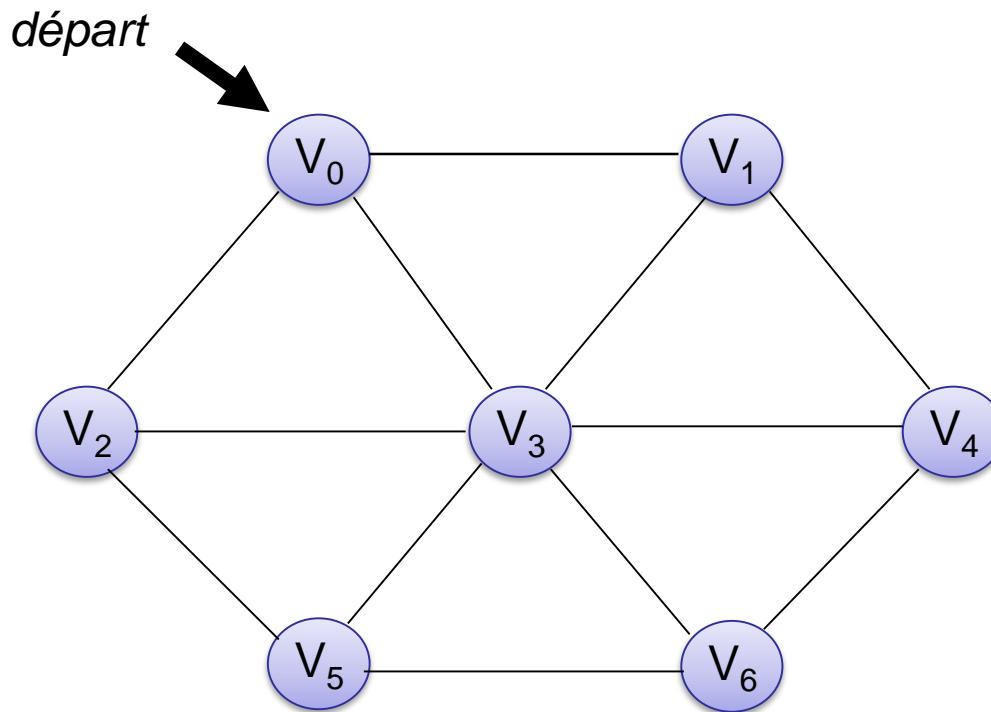
---



$V_0, V_1, V_3, V_2, V_5, V_6, V_4,$

# Ex. 3 DFS – graphe non orienté

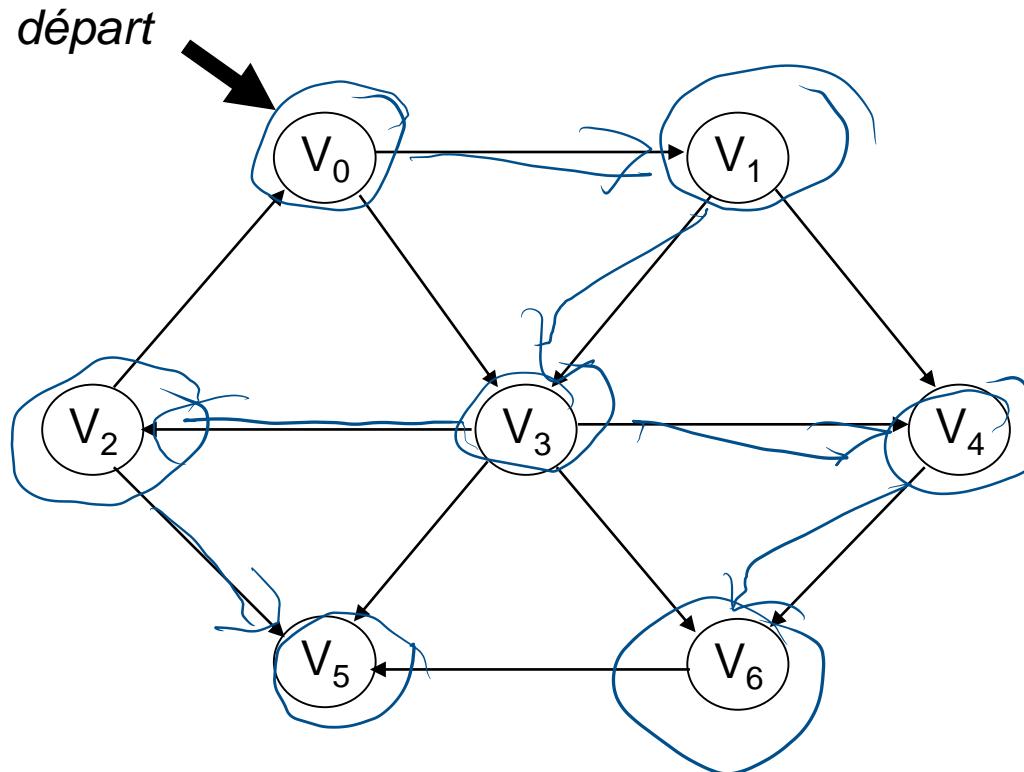
---



$V_0, V_1, V_3, V_2, V_5, V_6, V_4$ . FIN

# Ex. 4 DFS – graphe orienté

---



$V_0, V_1, V_3, V_2, V_5, V_4, V_6.$

---

# Graphes II



# Graphes II

---

1. Implementation
    1. Interface et classes de graphes orientés et non orientés
    2. Class Paths (BFS + DFS)
  2. Ordre topologique (version DFS)
    1. Parcours DFS post-ordre et post-ordre inverse
    2. Algorithme d'ordre topologique
  3. Composantes connexes
    1. Notion de connexité
    2. Composantes connexes (UG)
    3. Composantes fortement connexes (DG)
  4. Arbre sous-tendant minimum
    1. Problématique
    2. Algorithme de Prim
    3. Algorithme de Kruskal
-

# Graphes II

---

1. Implementation
    1. Interface et classes de graphes orientés et non orientés
    2. Class Paths (BFS + DFS)
  2. Ordre topologique (version DFS)
    1. Parcours DFS post-ordre et post-ordre inverse
    2. Algorithme d'ordre topologique
  3. Composantes connexes
    1. Notion de connexité
    2. Composantes connexes (UG)
    3. Composantes fortement connexes (DG)
  4. Arbre sous-tendant minimum
    1. Problématique
    2. Algorithme de Prim
    3. Algorithme de Kruskal
-

# Implémentation

---

- Un graphe implémentant Graph sera formé de sommets auxquels seront associés à des entiers allant de 0 à  $|V|-1$

```
import java.util.HashSet;

public interface Graph {
    void initialize(int V);
    int V(); // cardinal de l'ensemble des sommets
    int E(); // cardinal de l'ensemble des arcs
    void connect(int v1, int v2);
    HashSet<Integer> adj(int v); // liste d'adjacence
    String toString();
}
```

# Implémentation

---

- UndirectedGraph est un graphe non orienté sans poids sur les arcs implémentant Graph

```
import java.security.InvalidParameterException;
import java.util.HashSet;

public class UndirectedGraph implements Graph{

    private HashSet<Integer>[] neighbors; // listes d'adjacences
    private int V, E; // cardinal de V et cardinal de E

    public UndirectedGraph(int V){
        initialize(V);
    }
}
```

# Implémentation

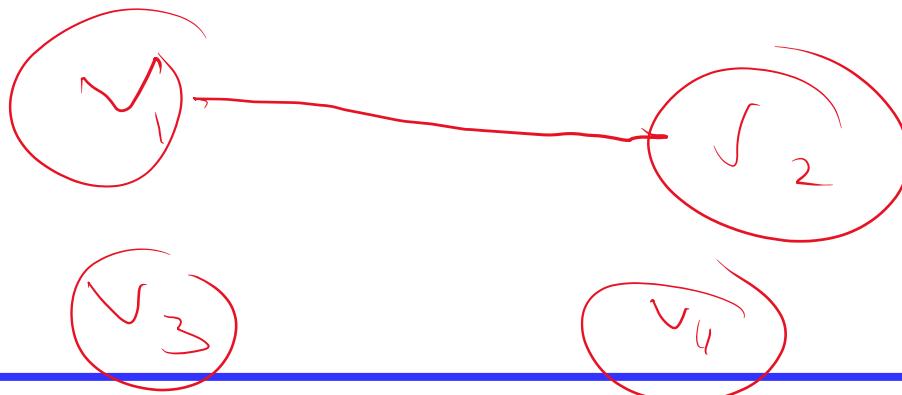
---

```
public void initialize(int V){  
    // check parameters  
    if(V < 0) throw new InvalidParameterException();  
  
    // initialize members  
    E = 0;  
    this.V = V;  
    neighbors = new HashSet[V];  
  
    for(int v=0; v<V; v++)  
        neighbors[v] = new HashSet<Integer>();  
}  
  
public int V(){return V;}  
public int E(){return E;}
```

# Implémentation

- Un graphe non orienté créera deux arcs pour relier deux sommets

```
public void connect(int v1, int v2){  
    // check parameters  
    if(v1<0 || v1>=V) return;  
    if(v2<0 || v2>=V) return;  
    if( neighbors[v1].contains(v2) ) return;  
  
    // connect in both directions  
    neighbors[v1].add(v2);  
    neighbors[v2].add(v1);  
    E++;  
}
```

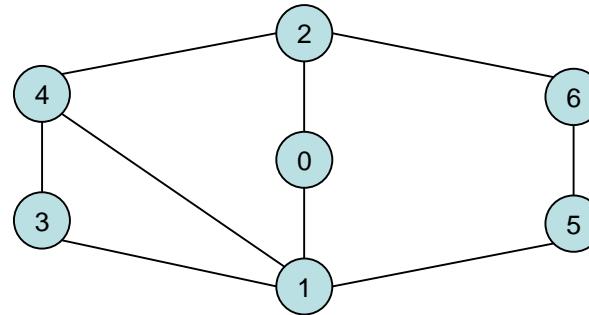


$$\begin{aligned} \overline{E} &= 0 \\ E &= 1 \end{aligned}$$

# Implémentation

- `toString()` nous servira à définir un graphe non orienté

```
public HashSet<Integer> adj(int v){  
    // check parameters  
    if(v<0 || v>=V) return null;  
    return neighbors[v];  
}  
  
public String toString(){  
    StringBuilder o = new StringBuilder();  
    String ln = System.getProperty("Line.separator");  
    o.append(V + ln + E + ln);  
    for(int v=0; v<V; v++)  
        for(int w : neighbors[v])  
            o.append(v + " - " + w + ln);  
    return o.toString();  
}
```



7  
9  
0-1  
0-2  
1-0  
1-3  
1-4  
1-5  
2-0  
2-4  
2-6  
3-1  
3-4  
4-1  
4-2  
4-3  
5-1  
5-6  
6-2  
6-5

# Implémentation

---

- `DirectedGraph` est un graphe orienté sans poids sur les arcs implémentant `Graph`

```
import java.security.InvalidParameterException;
import java.util.HashSet;

public class DirectedGraph implements Graph{

    private HashSet<Integer>[] neighbors; // listes d'adjacences
    private int V, E; // cardinal de V et cardinal de E

    public DirectedGraph(int V){
        initialize(V);
    }
}
```

# Implémentation

- Les méthodes `initialize(...)`, `v()` et `E()` sont identiques à celles de `UnirectedGraph`. `connect()` est également similaire, excepté qu' un seul arc est ajouté, il va de `v1` à `v2`

```
public void initialize(int V){...}
public int V(){return V;}
public int E(){return E;}

public void connect(int v1, int v2){
    // check parameters
    if(v1<0 || v1>=V) return;
    if(v2<0 || v2>=V) return;
    if( neighbors[v1].contains(v2) ) return;

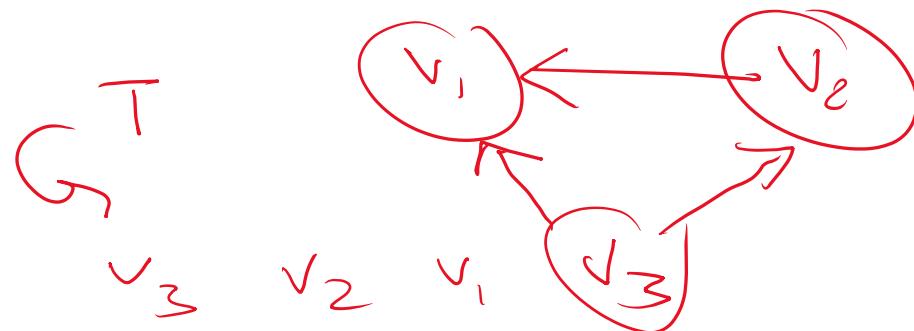
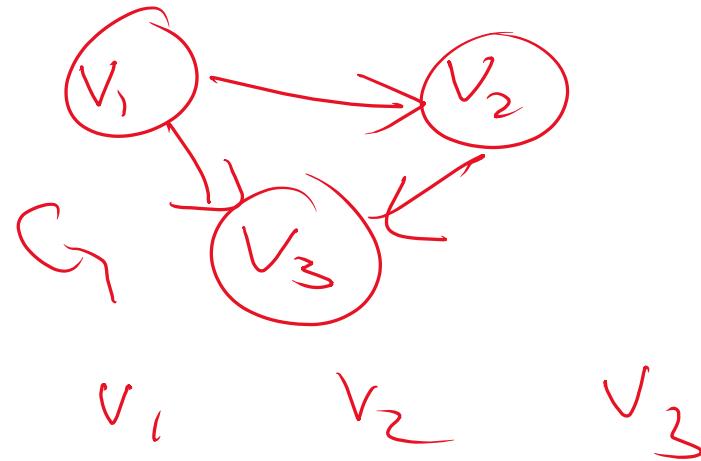
    // connect edge from v1 to v2
    neighbors[v1].add(v2); E++;
}
```



# Implémentation

- On ajoutera la méthode `transposed()` qui retourne un graphe orienté dont les arcs sont été inversés:

```
public DirectedGraph transposed(){  
    DirectedGraph T = new DirectedGraph(V);  
  
    for(int v=0; v<V; v++)  
        for(int w : neighbors[v])  
            T.connect(w, v);  
    return T;  
}
```

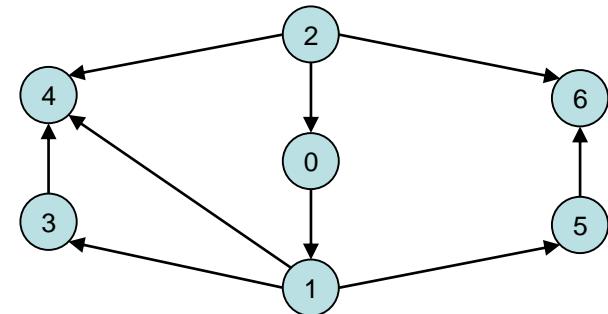


# Implémentation

- `toString()` nous servira à définir un graphe orienté (notez -> )

```
public HashSet<Integer> adj(int v){  
    // check parameters  
    if(v<0 || v>=V) return null;  
    return neighbors[v];  
}  
  
public String toString(){  
    StringBuilder o = new StringBuilder();  
    String ln = System.getProperty("line.separator");  
    o.append(V + ln + E + ln);  
    for(int v=0; v<V; v++)  
        for(int w : neighbors[v])  
            o.append(v + "->" + w + ln);  
    return o.toString();  
}
```

7  
9  
0->1  
1->3  
1->4  
1->5  
2->0  
2->4  
2->6  
3->4  
5->6



# Graphes II

- 
- 1. Implementation
    - 1. Interface et classes de graphes orientés et non orientés
    - 2. Class Paths (BFS + DFS)
  - 2. Ordre topologique (version DFS)
    - 1. Parcours DFS post-ordre et post-ordre inverse
    - 2. Algorithme d'ordre topologique
  - 3. Composantes connexes
    - 1. Notion de connexité
    - 2. Composantes connexes (UG)
    - 3. Composantes fortement connexes (DG)
  - 4. Arbre sous-tendant minimum
    - 1. Problématique
    - 2. Algorithme de Prim
    - 3. Algorithme de Kruskal

BFS

l'iveau  $\times$

file

DF

profondeur

réursive

# Implémentation

---

- Paths implémente les parcours de graphe

```
import java.security.InvalidParameterException;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class Paths {

    boolean[] dfsMarked, bfsMarked;
    int[] dfsParent, bfsParent;
    int s;
```

# Implémentation

---

- Le constructeur de Paths appelle les deux parcours

```
public Paths(Graph G, int s){  
    if(G == null || s < 0 || s>= G.V())  
        throw new InvalidParameterException();  
    this.s = s;  
  
    // process bfs  
    bfsMarked = new boolean[G.V()];  
    bfsParent = new int[G.V()];  
    bfs(G, s);  
  
    // process dfs  
    dfsMarked = new boolean[G.V()];  
    dfsParent = new int[G.V()];  
    dfs(G, s);  
}
```

# Implémentation

---

- BFS se fait au moyen d'une file:

```
private void bfs(Graph G, int s){  
    Queue<Integer> q = new LinkedList<Integer>();  
  
    // add source  
    q.add(s); bfsMarked[s] = true;  
  
    while( !q.isEmpty() ){  
        // poll vertex and treat neighbors  
        int v = q.poll();  
        for(int w : G.adj(v)){  
            if( !bfsMarked[w] ){  
                q.add(w);  
                bfsMarked[w] = true;  
                bfsParent[w] = v;  
            }  
        }  
    } // end while  
}
```

# Implémentation

---

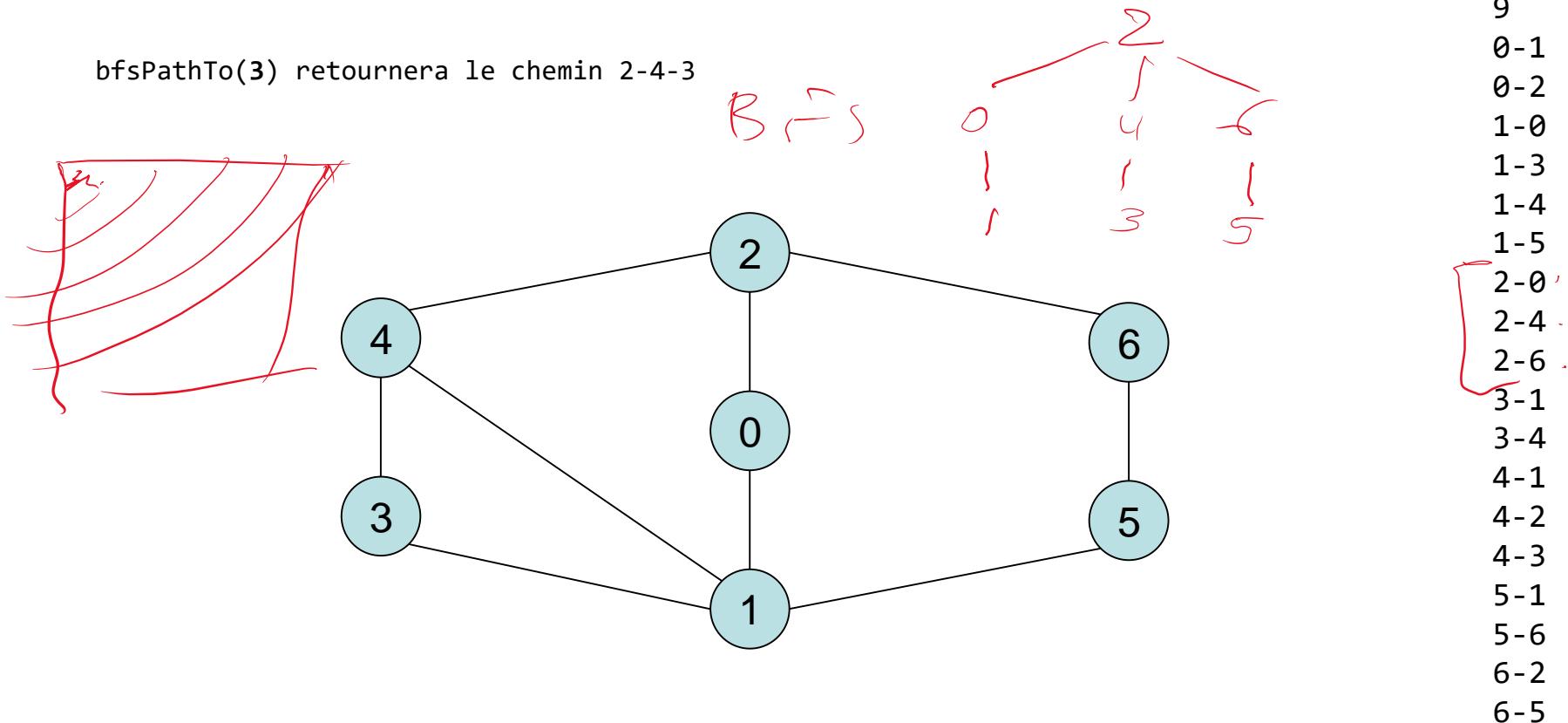
- On récupère le chemin BFS en empilant les parents depuis la destination jusqu'à la source:

```
public Stack<Integer> bfsPathTo(int v){  
  
    if( !bfsMarked[v] ) return null;  
  
    Stack<Integer> path = new Stack<Integer>();  
  
    for(int x = v; x != s; x = bfsParent[x])  
        path.push(x);  
    path.push(s);  
  
    return path;  
}  
}
```

# Implémentation

- Résultat sur le UndirectedGraph précédent ( $s == 2$ ):

bfsPathTo(3) retournera le chemin 2-4-3

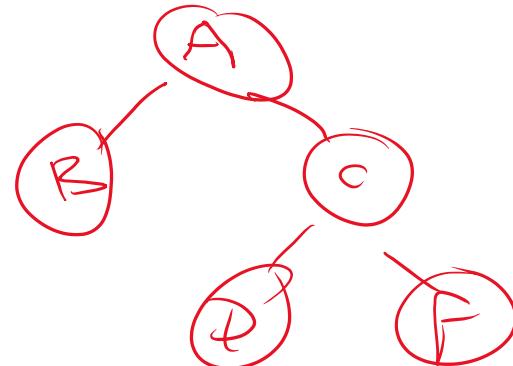


# Implémentation

- DFS se fait de manière récursive:

```
private void dfs(Graph G, int v){  
    dfsMarked[v] = true;  
  
    for(int w : G.adj(v))  
        if( !dfsMarked[w] ){  
            dfs(G, w);  
            dfsParent[w] = v;  
        }  
}
```

~~TINR~~  
LRN (Arbre)



B D F C A  
✓ ✓ ✓ ✓ ✓

# Implémentation

---

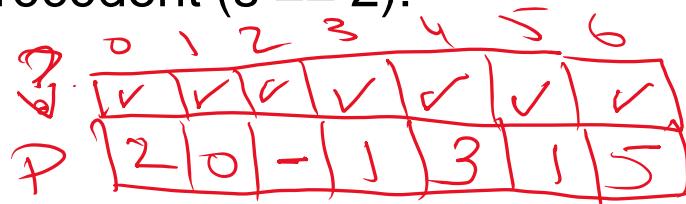
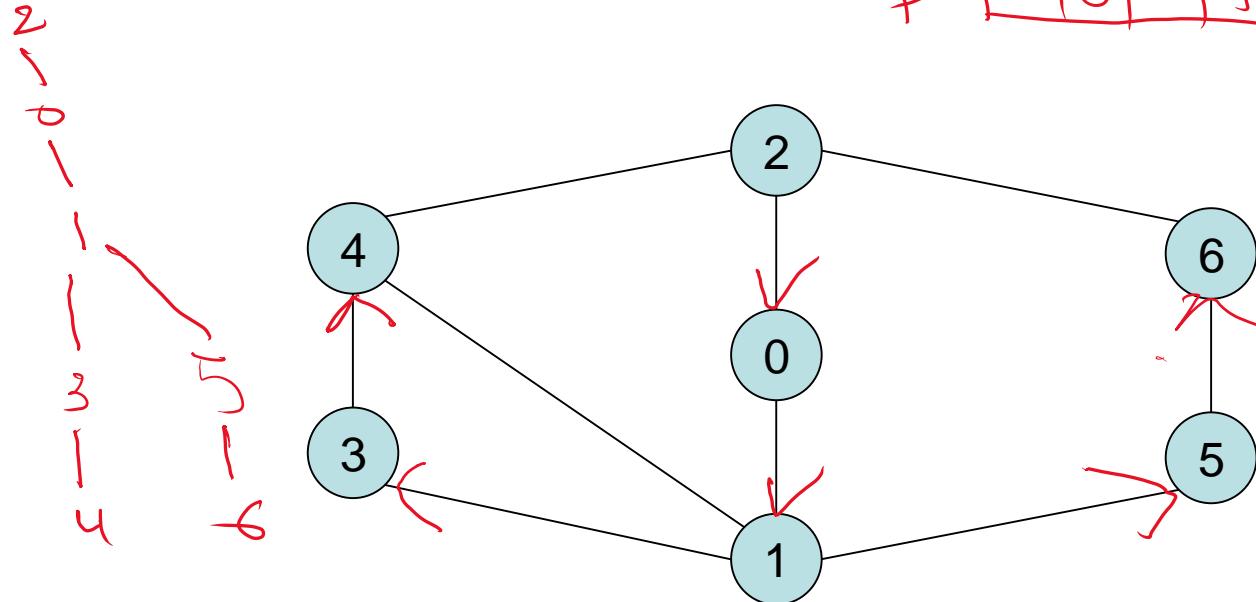
- On récupère le chemin DFS en empilant les parents depuis la destination jusqu'à la source:

```
public Stack<Integer> dfsPathTo(int v){  
  
    if( !dfsMarked[v] ) return null;  
  
    Stack<Integer> path = new Stack<Integer>();  
  
    for(int x = v; x != s; x = dfsParent[x])  
        path.push(x);  
    path.push(s);  
  
    return path;  
}
```

# Implémentation

- Résultat sur le UndirectedGraph précédent ( $s == 2$ ):

`dfsPathTo(3)` retournera le chemin 2-0-1-3



7  
9  
0-1  
0-2  
1-0  
1-3  
1-4  
1-5  
2-0  
2-4  
2-6  
3-1  
3-4  
4-1  
4-2  
4-3  
5-1  
5-6  
6-2  
6-5

# Graphes II

---

1. Implementation
  1. Interface et classes de graphes orientés et non orientés
  2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
  1. Parcours DFS post-ordre et post-ordre inverse
  2. Algorithme d'ordre topologique
3. Composantes connexes
  1. Notion de connexité
  2. Composantes connexes (UG)
  3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
  1. Problématique
  2. Algorithme de Prim
  3. Algorithme de Kruskal



# Ordre topologique II

---

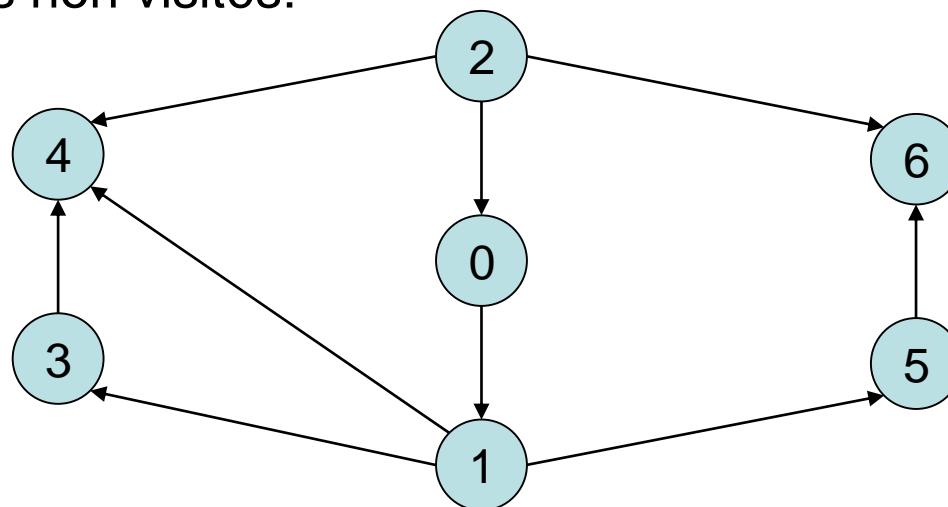
Rappel:

- Nous avons vu au cours précédent un algorithme permettant de déterminer l'ordre topologique d'un graphe dirigé acyclique
- L'algorithme du cours précédent se basait sur une file
- Nous allons voir un nouvel algorithme pour déterminer l'ordre topologique qui se base sur le parcours DFS 
- Pour ce faire, nous allons définir le parcours DFS post-ordre 

# Ordre topologique II

- Un parcours DFS post-ordre est le résultat d'un parcours en profondeur du graphe où un sommet est énuméré dès qu'il n'a plus de voisins non visités:

L R N



7  
9  
0->1  
1->3  
1->4  
1->5  
2->0  
2->4  
2->6  
3->4  
5->6

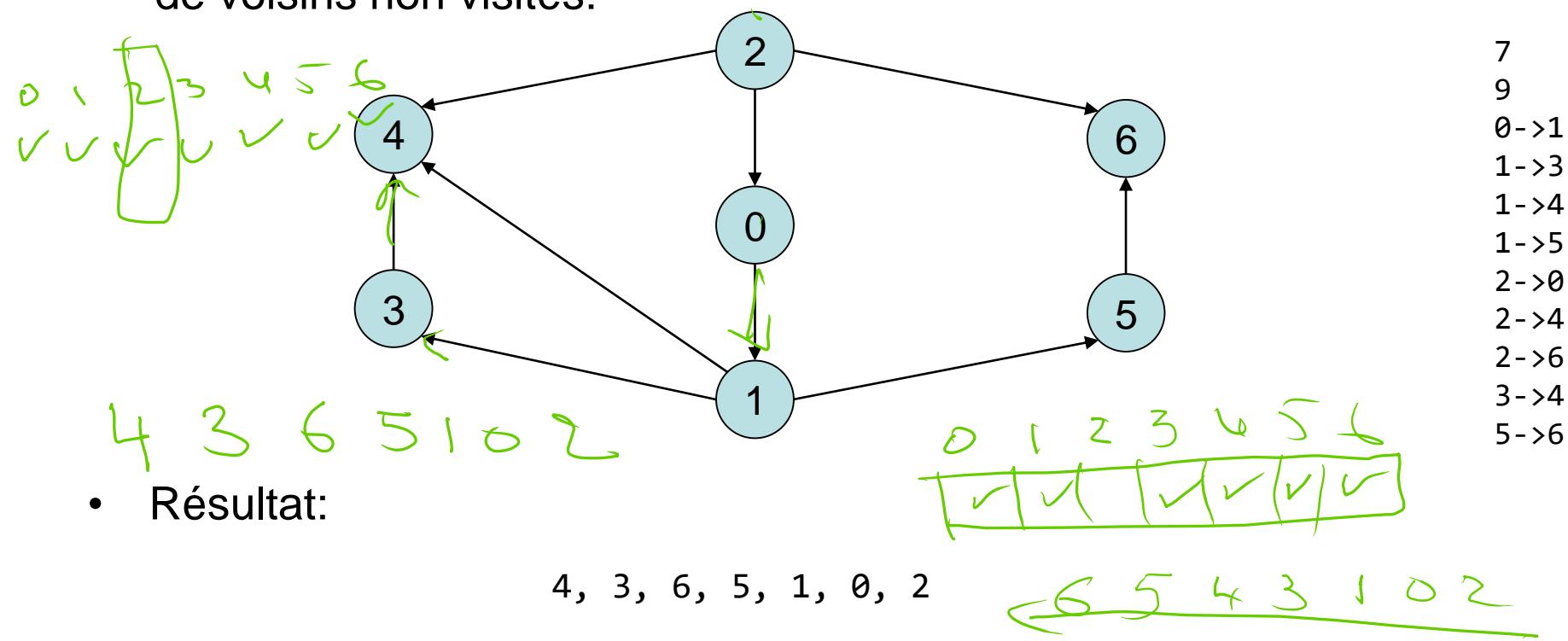
- Résultat:

4, 3, 6, 5, 1, 0, 2



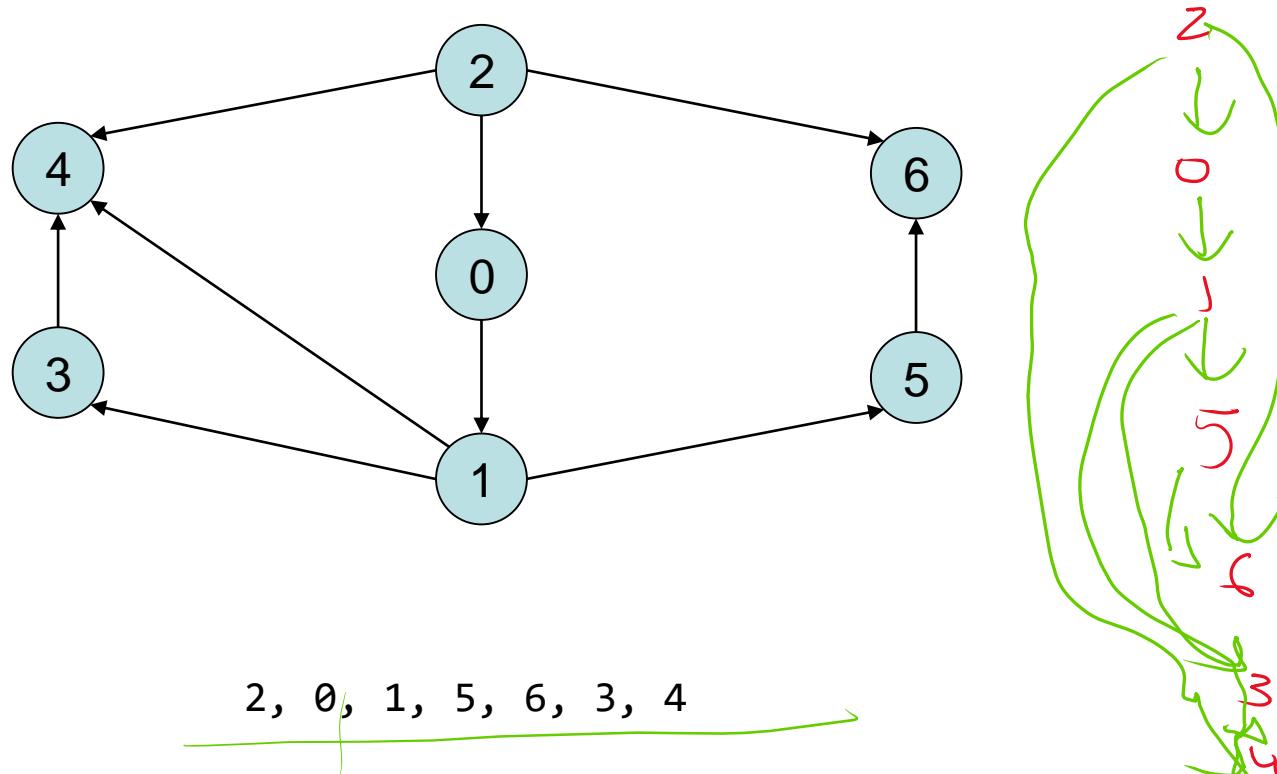
# Ordre topologique II

- Un parcours DFS post-ordre est le résultat d'un parcours en profondeur du graphe où un sommet est énuméré dès qu'il n'a plus de voisins non visités:



# Ordre topologique II

- Le parcours DFS post-ordre inverse est le résultat inverse du parcours DFS post-ordre:



- Résultat:

# Ordre topologique II

---

- On modifie DFS comme suit:

```
// new Paths member, must be initialized in constructor
private Stack<Integer> reversePostOrderDfs;

private void dfs(Graph G, int v){
    dfsMarked[v] = true;

    for(int w : G.adj(v))
        if( !dfsMarked[w] ){
            dfs(G, w);
            dfsParent[w] = v;
        }

    // Stack vertex
    reversePostOrderDfs.push(v);
}
```



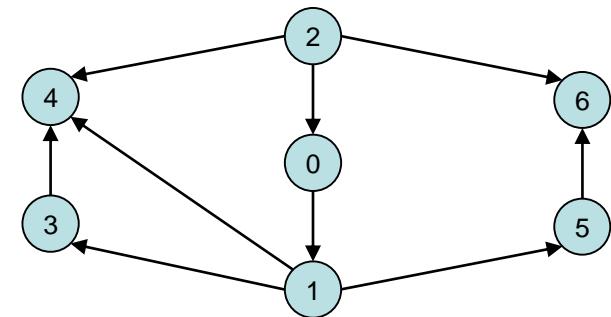
- L'ordre des nœuds du graphe obtenu d'un DFS post-ordre inverse est un ordre topologique:
-

# Ordre topologique II

- Comparons avec l'algorithme vu au cours 10:

DFS Post-ordre 2, 0, 1, 5, 6, 3, 4

	Indegree						
0	1	0	-	-	-	-	-
1	1	1	0	-	-	-	-
2	0	-	-	-	-	-	-
3	1	1	1	0	-	-	-
4	3	2	2	1	0	-	-
5	1	1	1	0	-	-	-
6	2	1	1	4	1	0	-
Entre en file	2	0	1	3,5	4	6	-
Sort de file	2	0	1	3	5	4	6



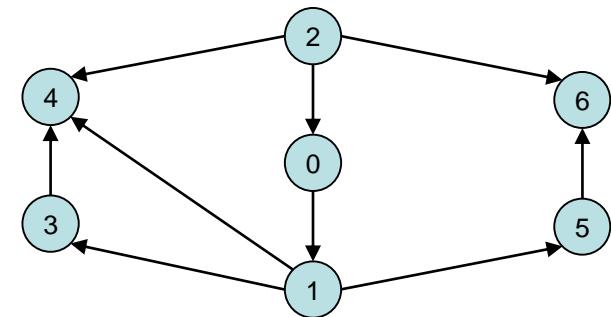
# Ordre topologique II

- Comparons avec l'algorithme vu au cours 10:

DFS Post-ordre 2, 0, 1, 5, 6, 3, 4



	Indegree							
0	1	0	-	-	-	-	-	
1	1	1	0	-	-	-	-	
2	0	-	-	-	-	-	-	
3	1	1	1	0	-	-	-	
4	3	2	2	1	0	-	-	
5	1	1	1	0	-	-	-	
6	2	1	1	1	1	0	-	
Entre en file	2	0	1	3, 5	4	6	-	
Sort de file	2	0	1	3	5	4	6	



# Graphes II

---

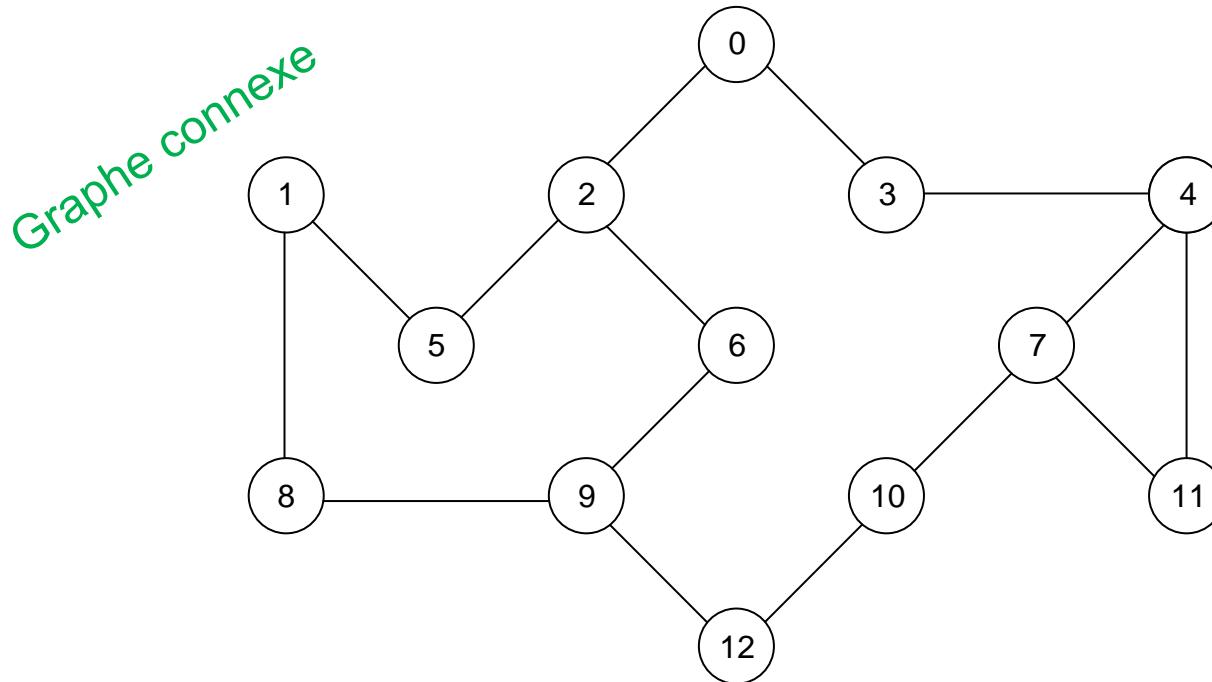
1. Implementation
  1. Interface et classes de graphes orientés et non orientés
  2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
  1. Parcours DFS post-ordre et post-ordre inverse
  2. Algorithme d'ordre topologique
3. Composantes connexes
  1. Notion de connexité
  2. Composantes connexes (UG)
  3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
  1. Problématique
  2. Algorithme de Prim
  3. Algorithme de Kruskal

# Composantes connexes

---

Rappel:

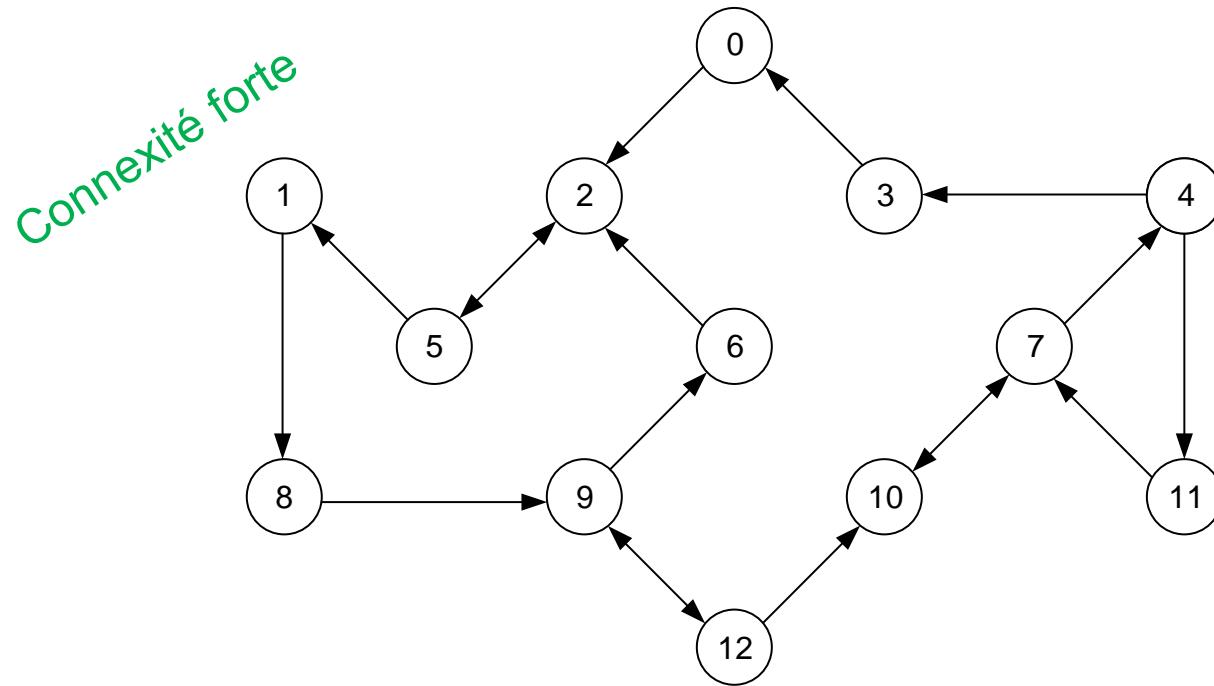
- Graphe connexe → un chemin pour chaque paire de nœuds



# Composantes connexes

Rappel:

- Si un graphe orienté est connexe → on dit qu'il a une connexité forte



# Graphes II

---

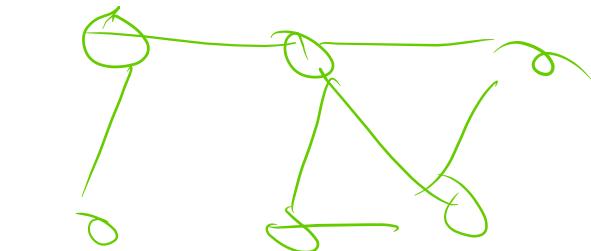
1. Implementation
  1. Interface et classes de graphes orientés et non orientés
  2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
  1. Parcours DFS post-ordre et post-ordre inverse
  2. Algorithme d'ordre topologique
3. Composantes connexes
  1. Notion de connexité
  2. **Composantes connexes (UG)**
  3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
  1. Problématique
  2. Algorithme de Prim
  3. Algorithme de Kruskal



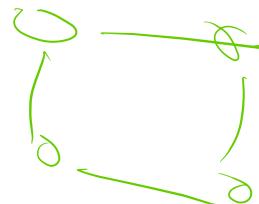
# Composantes connexes

Objectif:

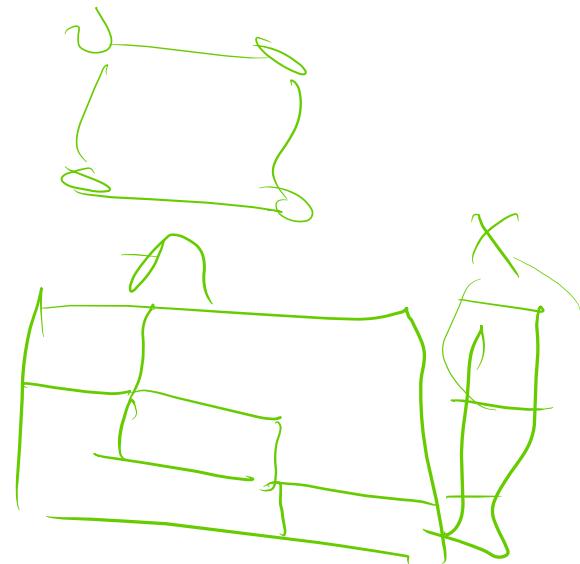
- Dans un graphe non orienté, identifier les composantes connexes.  
Par définition, un graphe connexe ne possèdera qu'une seule composante connexe.



$$\begin{aligned} A_1 x_1 &= b_1 \\ A_2 x_2 &= b_2 \end{aligned}$$



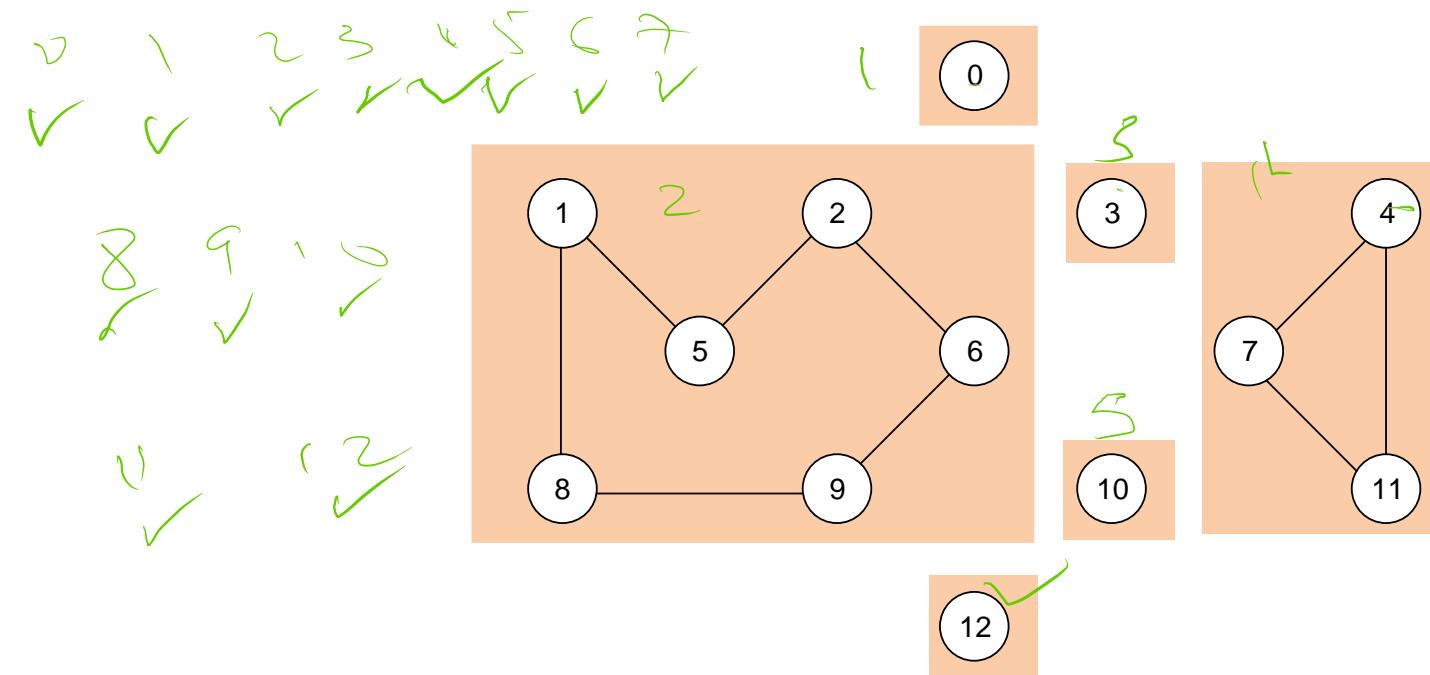
$$\begin{aligned} A_1 x_1 &= b_1 \\ A_2 x_2 &= b_2 \\ A_3 x_3 &= b_3 \end{aligned}$$



# Composantes connexes

Exemples:

- Ce graphe non dirigé possède 6 composantes connexes



# Composantes connexes

## Solution:

- Il suffit d'exécuter un parcours DFS. À chaque interruption, on débute une nouvelle composante connexe.

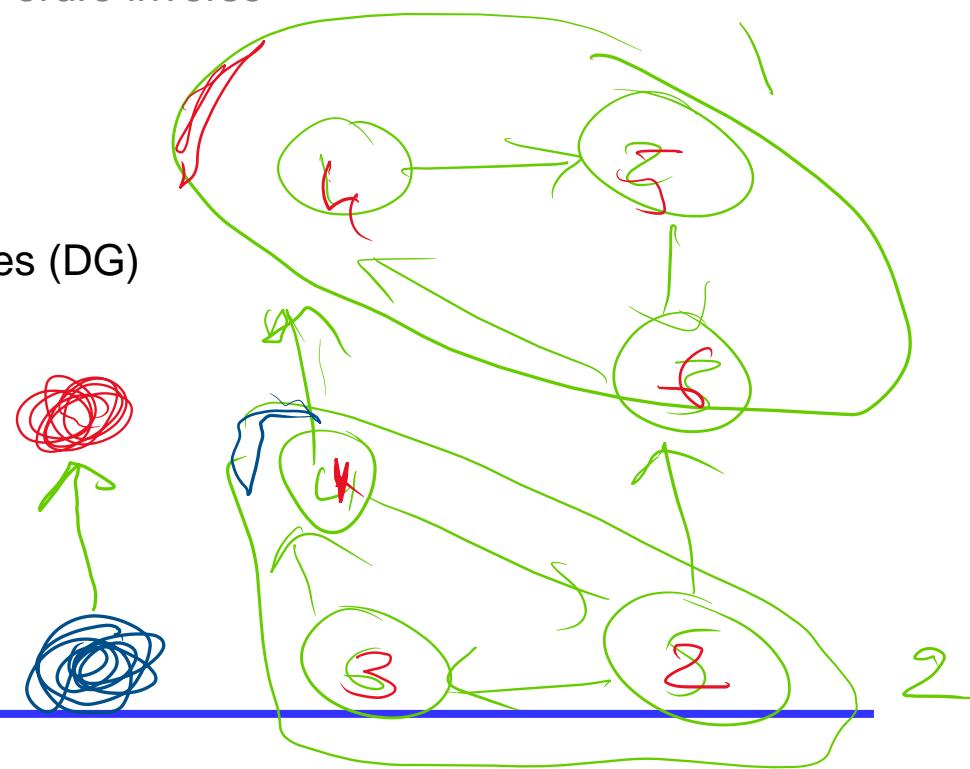
```
public class ConnectedComponents {  
    private boolean[] marked;  
    private int[] id;  
    private int count;  
  
    public ConnectedComponents(UndirectedGraph G){  
        if(G == null) throw new InvalidParameterException();  
  
        marked = new boolean[G.V()];  
        id     = new int[G.V()];  
  
        for(int v=0; v<G.V(); v++){  
            if( !marked[v] ){  
                dfs(v, G);  
                count++; // new component  
            }  
        }  
    }  
}
```

```
private void dfs(int v, Graph G){  
    marked[v] = true;  
  
    // identify component  
    id[v] = count;  
  
    for(int w : G.adj(v))  
        if(!marked[w])  
            dfs(w, G);  
}
```

1 2 3 4 5 6

# Graphes II

- 
1. Implementation
    1. Interface et classes de graphes orientés et non orientés
    2. Class Paths (BFS + DFS)
  2. Ordre topologique (version DFS)
    1. Parcours DFS post-ordre et post-ordre inverse
    2. Algorithme d'ordre topologique
  3. Composantes connexes
    1. Notion de connexité
    2. Composantes connexes (UG)
    3. Composantes fortement connexes (DG)
  4. Arbre sous-tendant minimum
    1. Problématique
    2. Algorithme de Prim
    3. Algorithme de Kruskal



# Composantes fortement connexes

---

## Objectif:

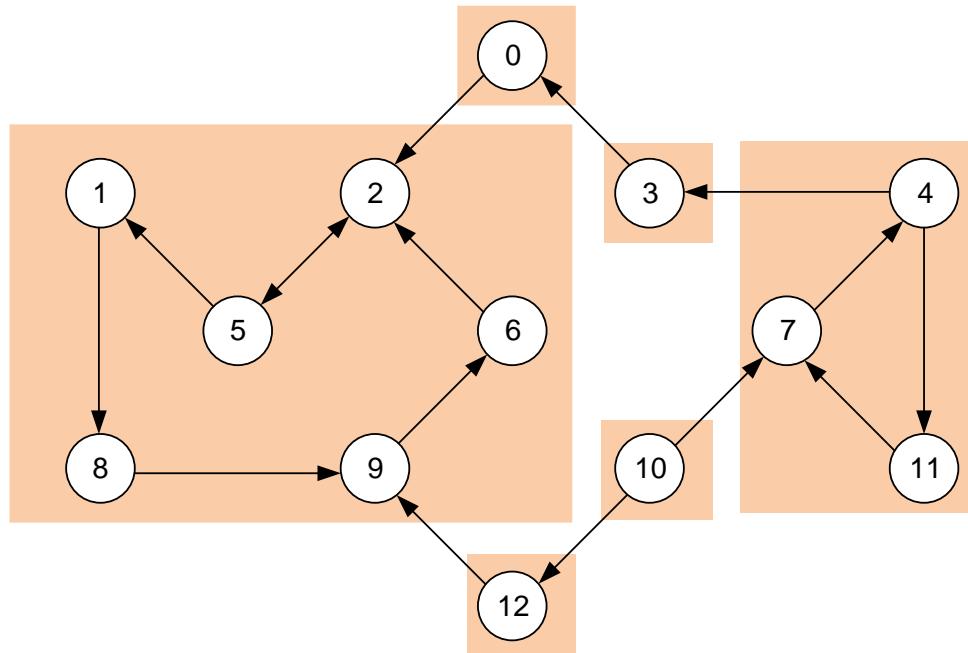
- Dans un graphe orienté, identifier les composantes fortement connexes. Par définition, un graphe orienté fortement connexe ne possèdera qu'une seule composante connexe.

# Composantes fortement connexes

---

Exemples:

- Ce graphe orienté possède également 6 composantes fortement connexes

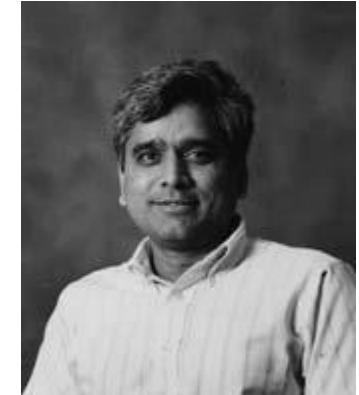


# Composantes fortement connexes

---

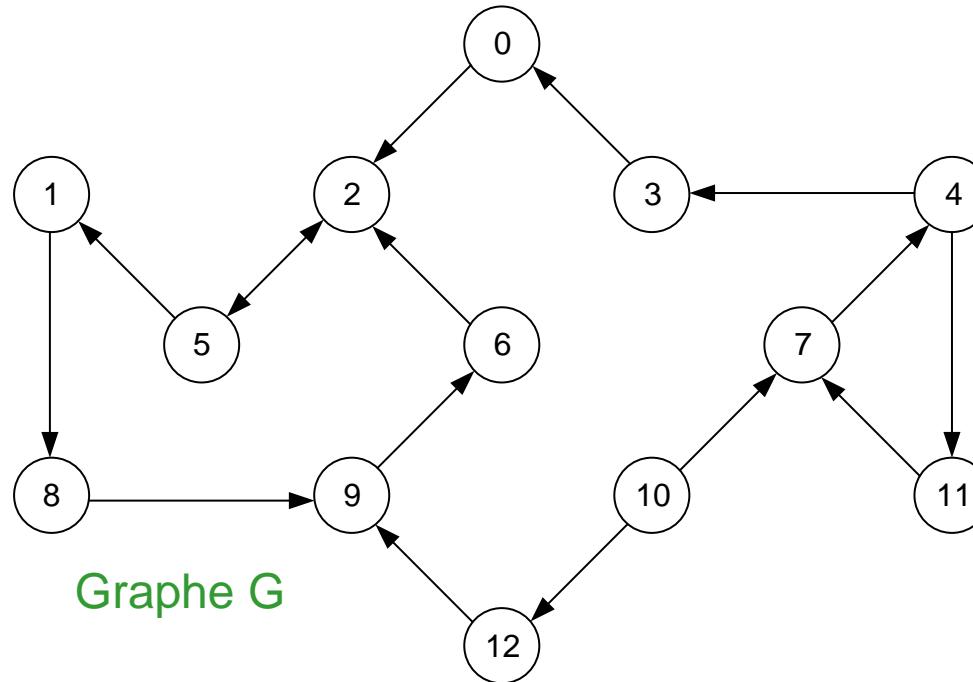
## Solution:

- Évidemment, un parcours DFS ne suffit pas.
- On remarquera cependant que les composantes fortement connexes de  $G$  le sont également de  $G^T$ .
- Un algorithme se basant sur cette observation et dû à S. Rao Kosaraju permet de résoudre le problème



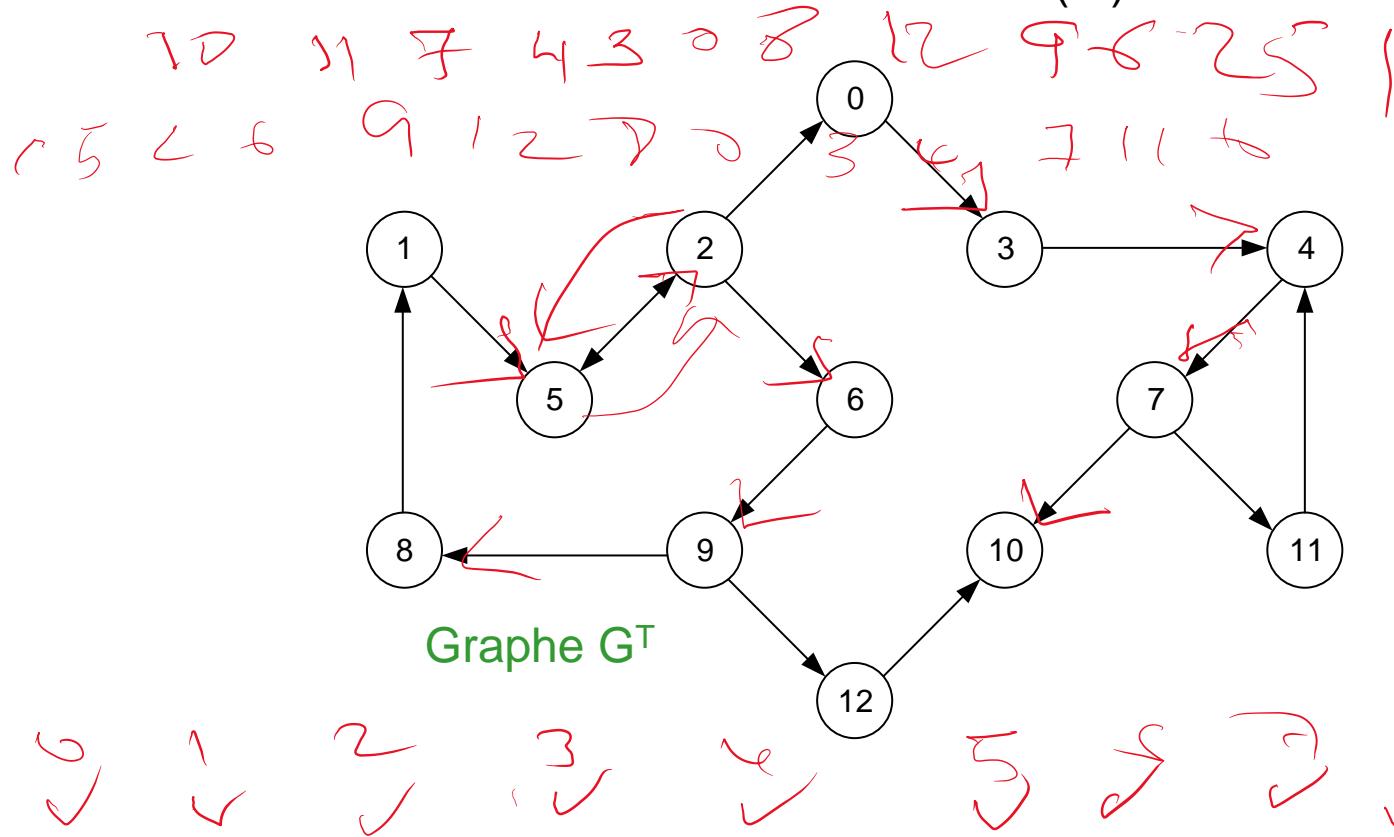
# Composantes fortement connexes

1. Ordonner les nœuds du graphe obtenus d'un DFS post-ordre inverse de  $G^T$
2. Parcourir  $G$  en DFS suivant l'ordre obtenu en (1.)



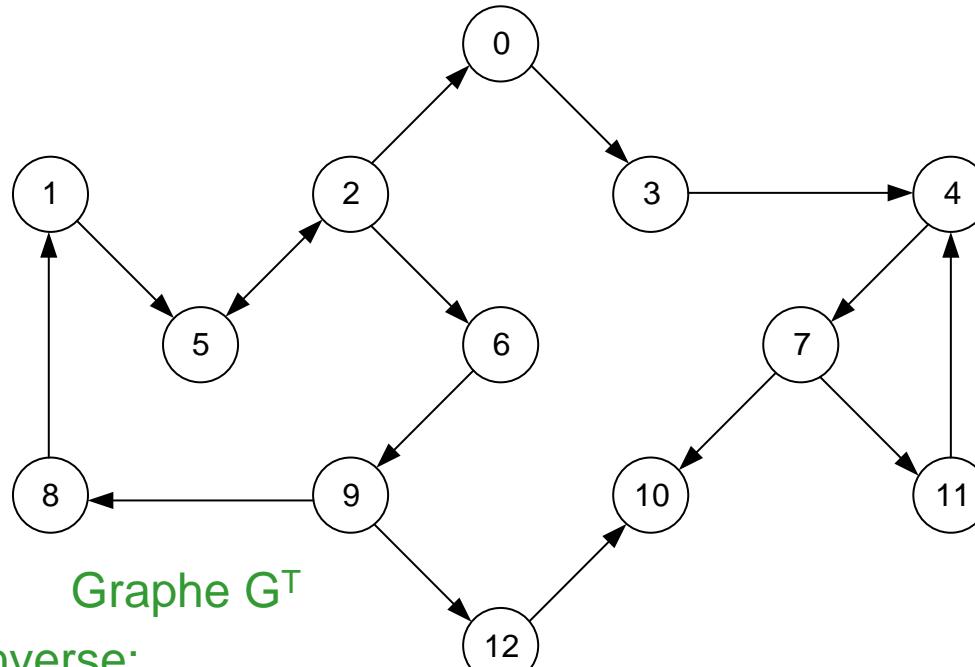
# Composantes fortement connexes

1. Ordonner les nœuds du graphe obtenus d'un DFS post-ordre inverse de  $G^T$
2. Parcourir  $G$  en DFS suivant l'ordre obtenu en (1.)



# Composantes fortement connexes

1. **Ordonner les nœuds du graphe obtenus d'un DFS post-ordre inverse de  $G^T$**
2. Parcourir  $G$  en DFS suivant l'ordre obtenu en (1.)

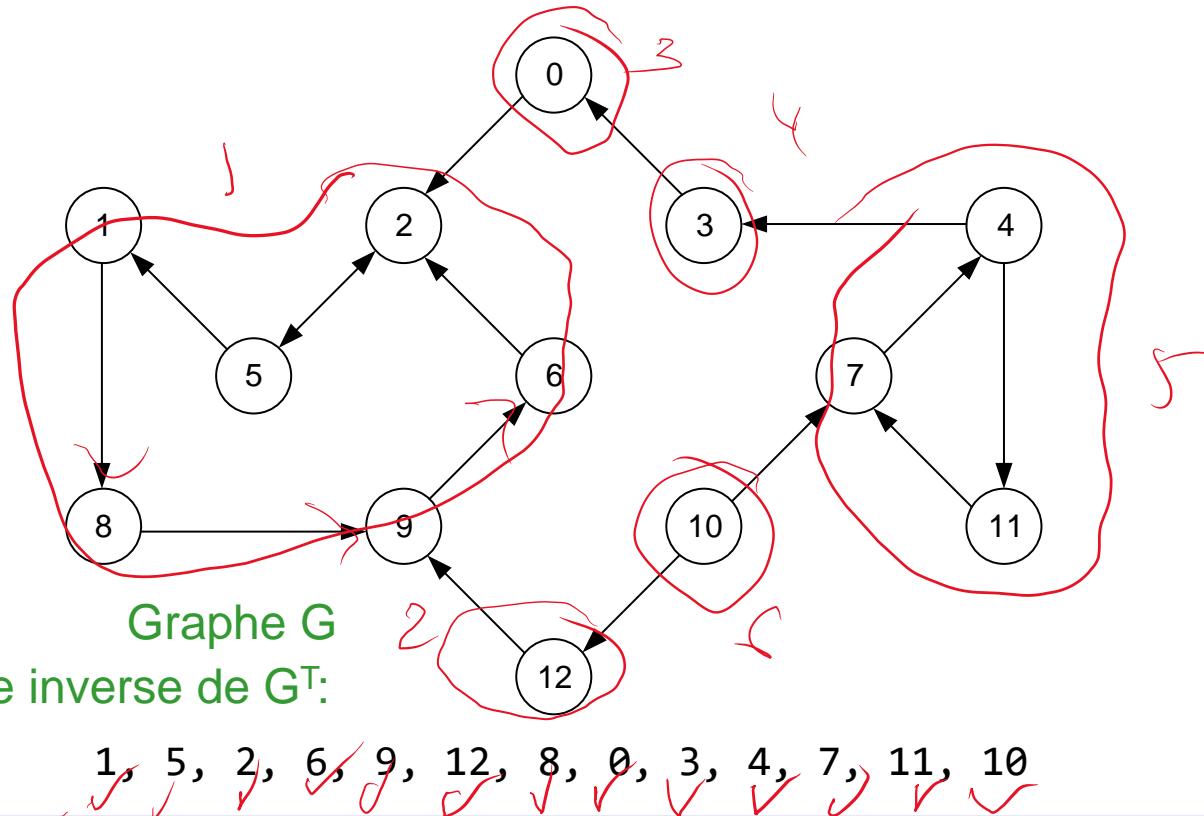


Post-ordre inverse:

1, 5, 2, 6, 9, 12, 8, 0, 3, 4, 7, 11, 10

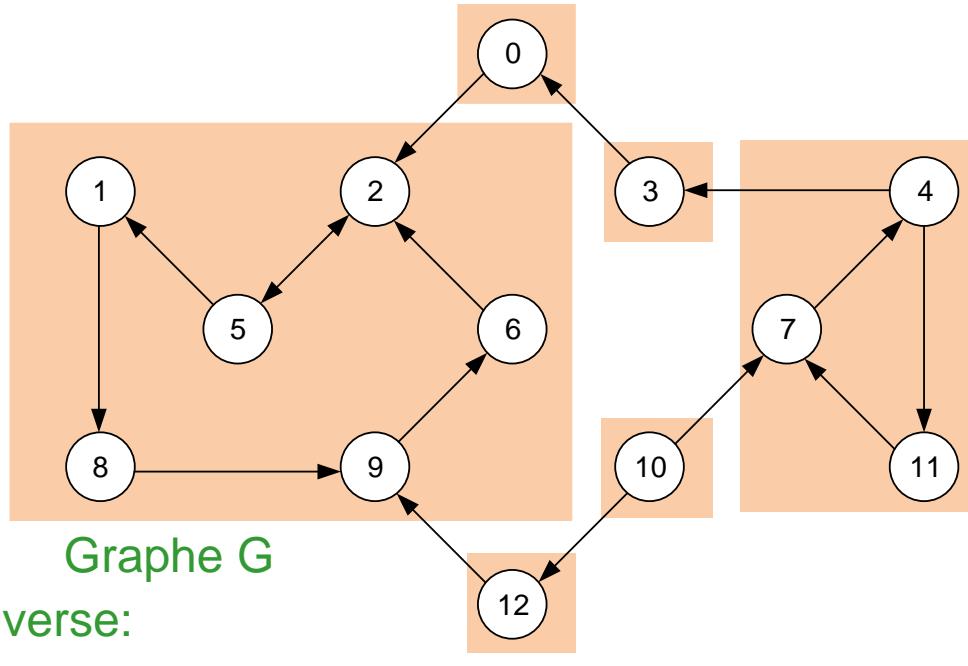
# Composantes fortement connexes

1. Ordonner les nœuds du graphe obtenus d'un DFS post-ordre inverse de  $G^T$
2. Parcourir  $G$  en DFS suivant l'ordre obtenu en (1.)



# Composantes fortement connexes

1. **Ordonner les nœuds du graphe obtenus d'un DFS post-ordre inverse de  $G^T$**
2. Parcourir  $G$  en DFS suivant l'ordre obtenu en (1.)



# Composantes fortement connexes

---

- Algorithme de KosarajuSharir.

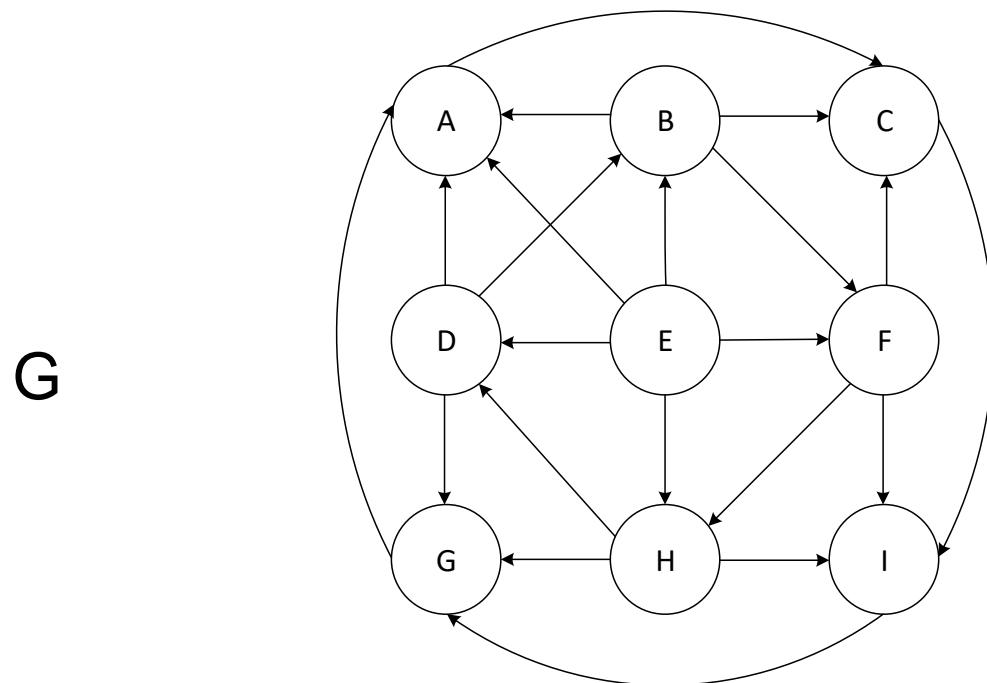
```
public class StrongConnectedComponents {  
    private boolean[] marked;  
    private int[] id;  
    private int count;  
  
    public StrongConnectedComponents(DirectedGraph G){  
        if(G == null) throw new InvalidParameterException();  
  
        DepthFirstOrder dfo = new DepthFirstOrder(G.transposed());  
  
        marked = new boolean[G.V()];  
        id     = new int[G.V()];  
  
        for( int v=0 : dfo.reversePost() )  
            if( !marked[v] ){  
                dfs(v, G);  
                count++; // new component  
            }  
    }  
}
```

```
private void dfs(int v, Graph G){  
    marked[v] = true;  
  
    // identify component  
    id[v] = count;  
  
    for(int w : G.adj(v))  
        if(!marked[w])  
            dfs(w, G);  
}
```

# Composantes fortement connexes

---

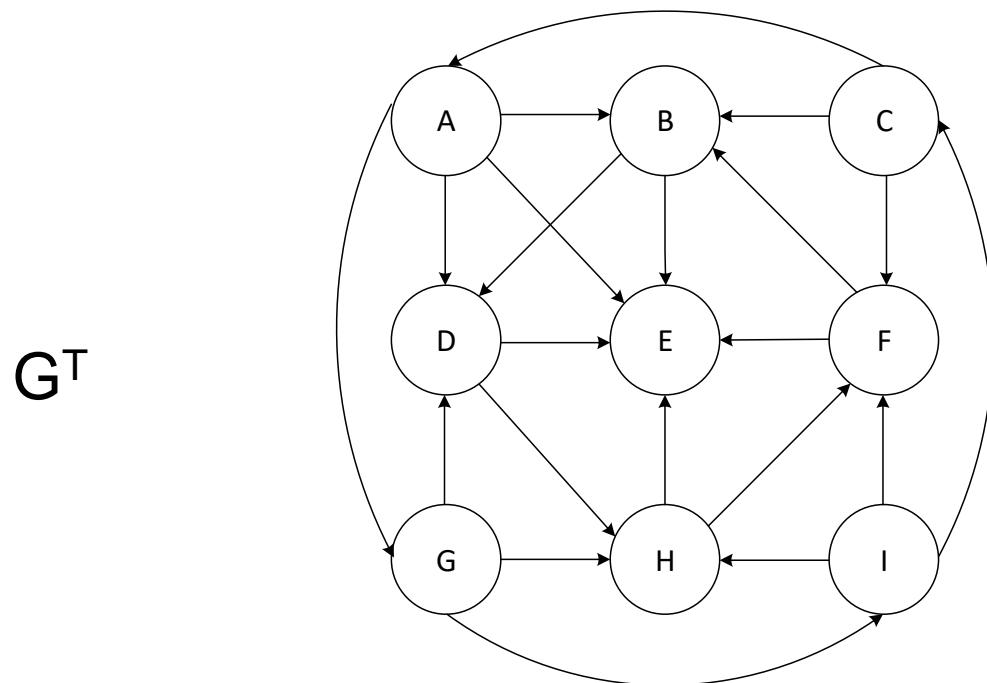
- Exemple



# Composantes fortement connexes

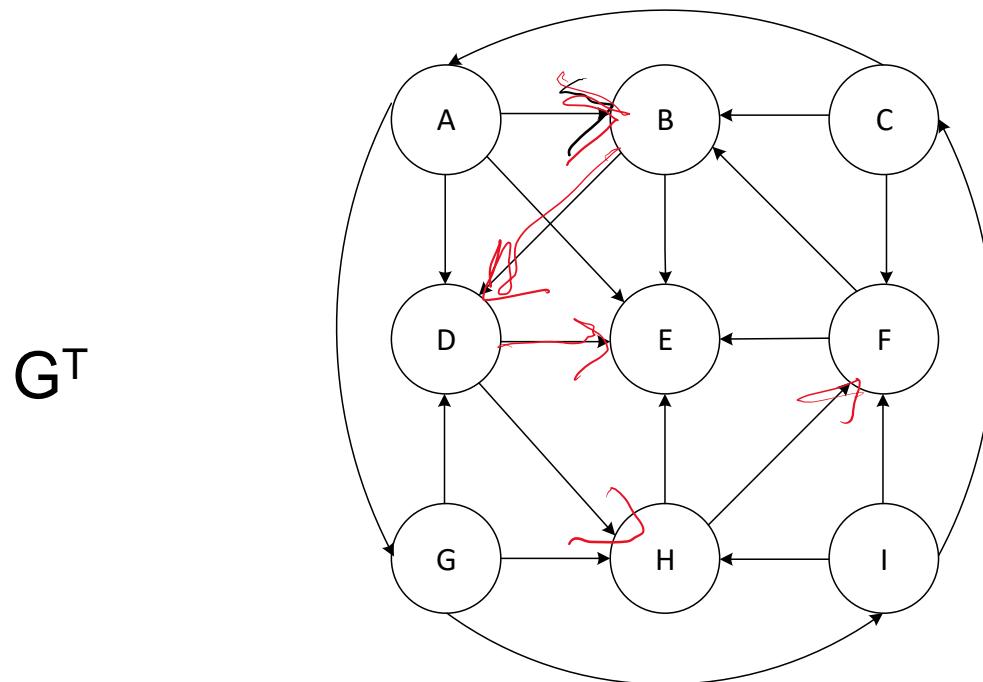
---

- Exemple



# Composantes fortement connexes

- Exemple

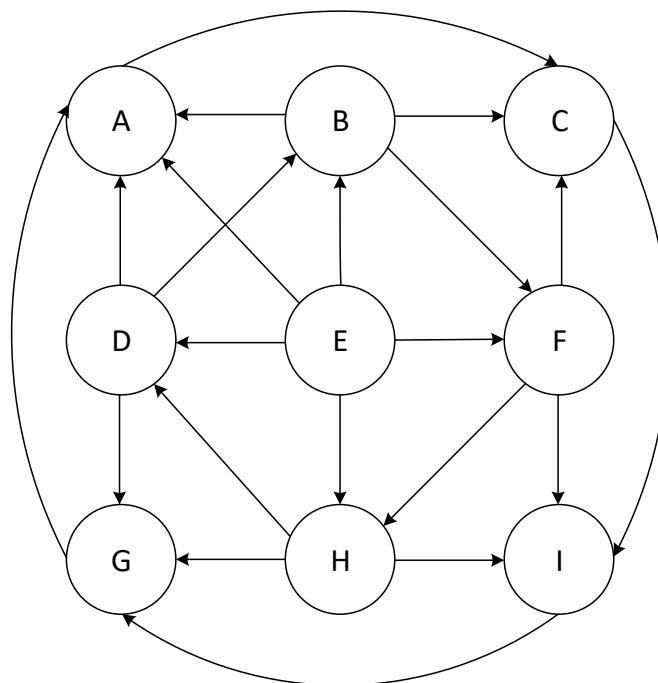


A B C D E F G H I  
X X X X X X X X X  
- A I B D H A D G  
DFS P.O.: E F H D B C I G A  
DFS P.O.I.: A G I C B D H F E

# Composantes fortement connexes

- Exemple

G



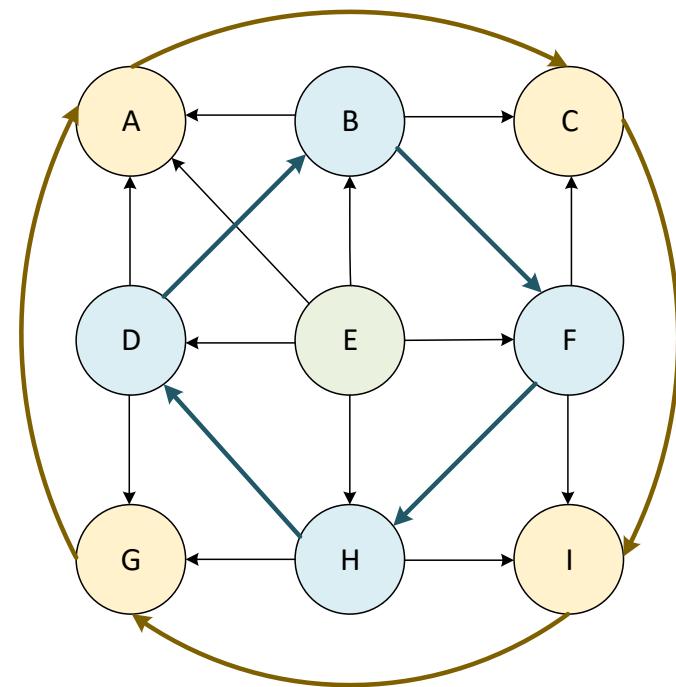
DFS P.O.I.: A G I C B D H F E

A G I C B D H F E  
X X X X X X X X X  
- I C A - H F B -  
0 0 0 0 1 1 1 1 2

# Composantes fortement connexes

- Exemple

G

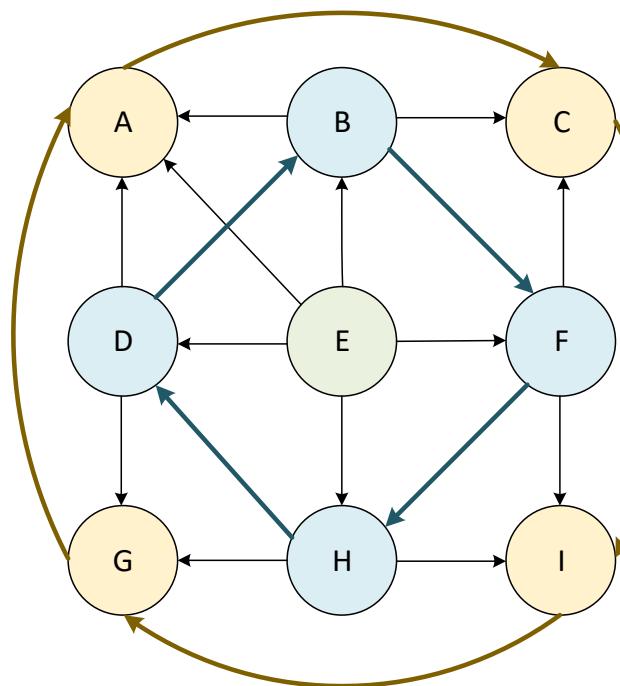
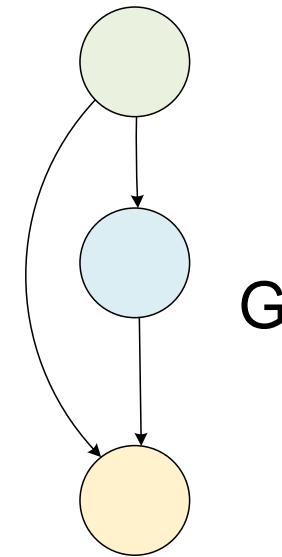


DFS P.O.I.: A G I C B D H F E

A G I C B D H F E  
X X X X X X X X X  
- I C A - H F B -  
0 0 0 0 1 1 1 1 2

# Composantes fortement connexes

- Exemple



**DFS P.O.I.: A G I C B D H F E**

A	G	I	C	B	D	H	F	E
X	X	X	X	X	X	X	X	X
-	I	C	A	-	H	F	B	-
0	0	0	0	1	1	1	1	2

DFS P.O.: E F H D B C I G A

# Graphes II

---

1. Implementation
  1. Interface et classes de graphes orientés et non orientés
  2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
  1. Parcours DFS post-ordre et post-ordre inverse
  2. Algorithme d'ordre topologique
3. Composantes connexes
  1. Notion de connexité
  2. Composantes connexes (UG)
  3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
  1. Problématique
  2. Algorithme de Prim
  3. Algorithme de Kruskal

# Arbre sous-tendant minimum

---

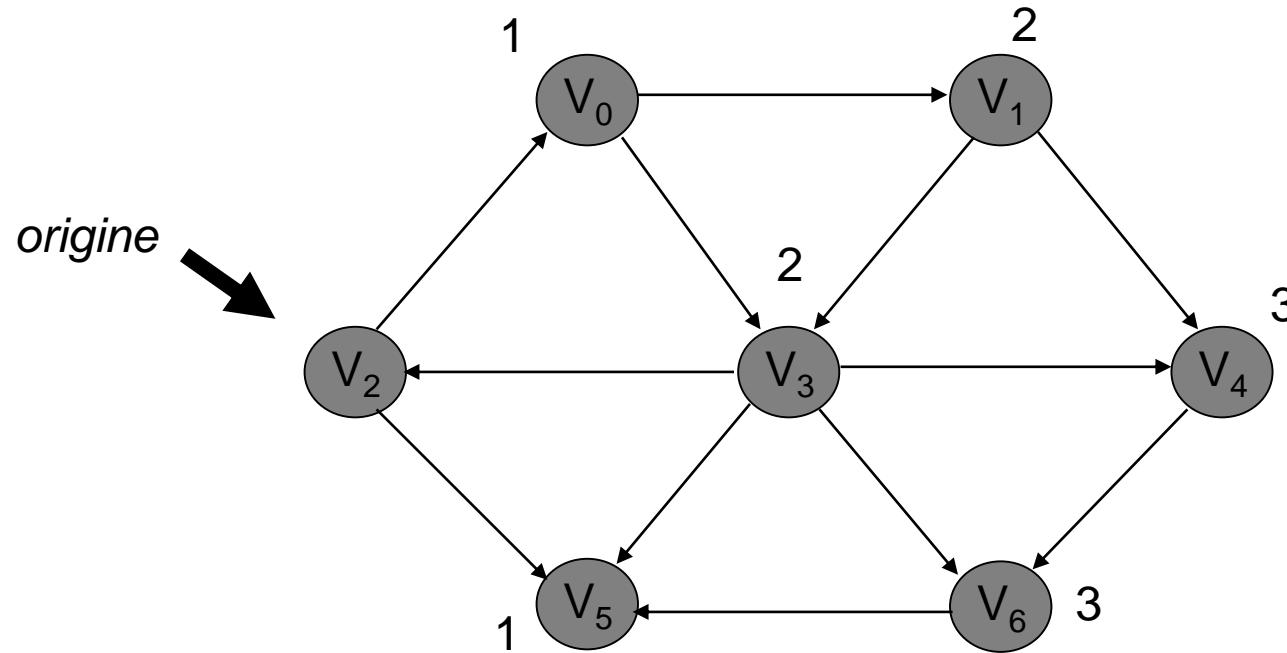
## Problématique:

On cherche à relier toutes les sommets d'un graphe valué non orienté en ne retenant que certaines de ses arêtes, de sorte à réduire le coût total associé aux arêtes choisies.

Ce faisant, on définit un arbre sous-tendant le graphe. Cet arbre sous-tendant est dit minimum car le coût qui lui est associé est le plus bas sur l'ensemble des arbres sous-tendant ledit graphe.

# Arbre sous-tendant minimum

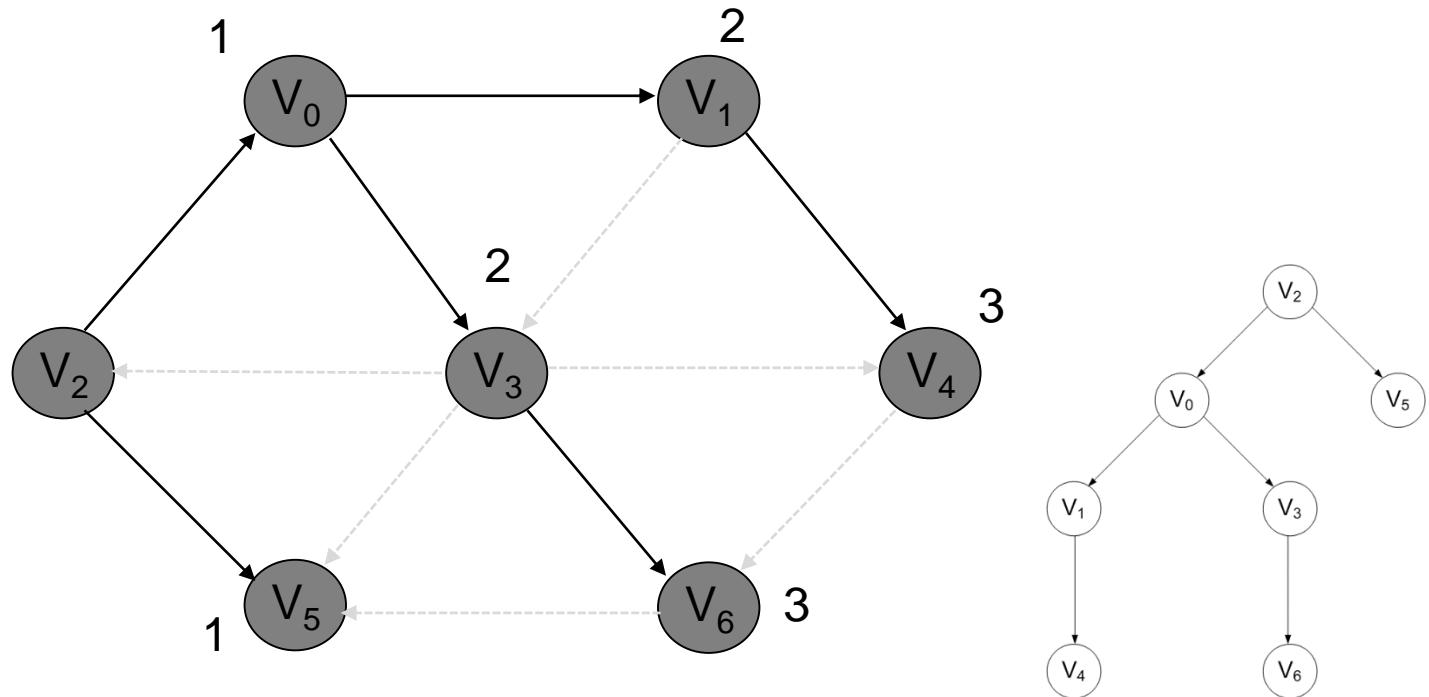
Rappel: Nous avions vu le rapprochement entre l'algorithme de chemin le plus court exécuté sur un graphe non valué:



File: vide

# Arbre sous-tendant minimum

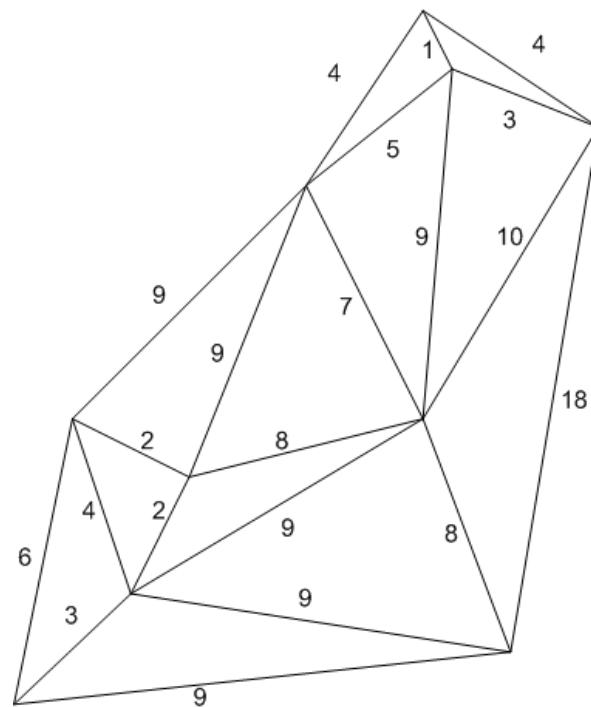
Le concept d'arbre sous-tendant d'un graphe n'est pas différent: il consiste à créer un arbre depuis un graphe en soustrayant certaines arêtes.



# Arbre sous-tendant minimum

---

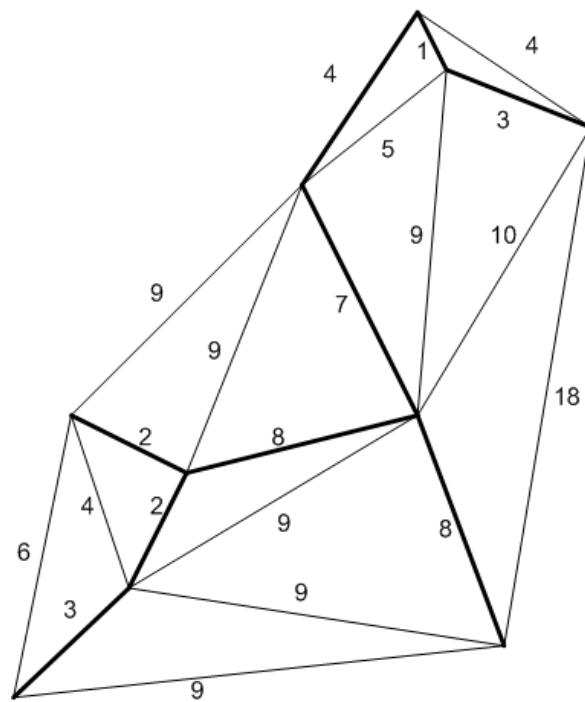
Le concept d'arbre sous-tendant **minimum** s'applique à un arbre valué non dirigé:



# Arbre sous-tendant minimum

---

dont on veut **minimiser** le coût:



# Arbre sous-tendant minimum

## **Cas type d'application – réseau de communication:**



Fig. 1 — Example of a shortest connection network.

**Extrait de :** R. C. Prim: *Shortest connection networks and some generalizations* In: *Bell System Technical Journal*, 36 (1957), pp. 1389–1401

# Graphes II

---

1. Implementation
  1. Interface et classes de graphes orientés et non orientés
  2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
  1. Parcours DFS post-ordre et post-ordre inverse
  2. Algorithme d'ordre topologique
3. Composantes connexes
  1. Notion de connexité
  2. Composantes connexes (UG)
  3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
  1. Problématique
  2. **Algorithme de Prim**
  3. Algorithme de Kruskal



# Algorithme de Prim

L'algorithme que nous allons voir est dû à Robert Prim.

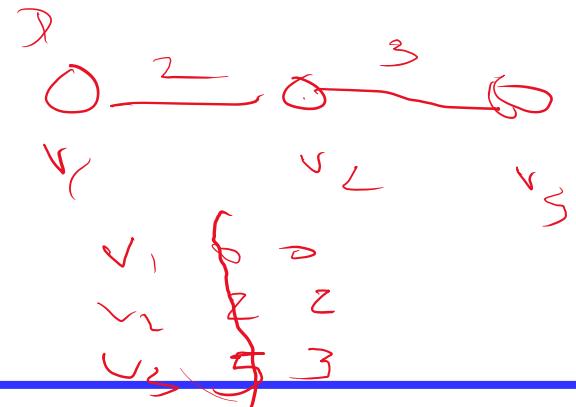
Ce dernier l'a publié en 1957. Il s'agit d'un algorithme **glouton**: un choix optimal est réalisé étape par étape, jusqu'à obtenir la solution:

L'algorithme de Prim a le même comportement que Dijkstra, à quelques différences près.

On maintient les informations suivantes pour chaque nœud:

- 1.La distance de l'arête arrivant sur  $v$  depuis le sommet parent ( $d_v$ );
- 2.Un booléen informant si le sommet est connu
- 3.Le parent à date du sommet  $v$  ( $p_v$ )

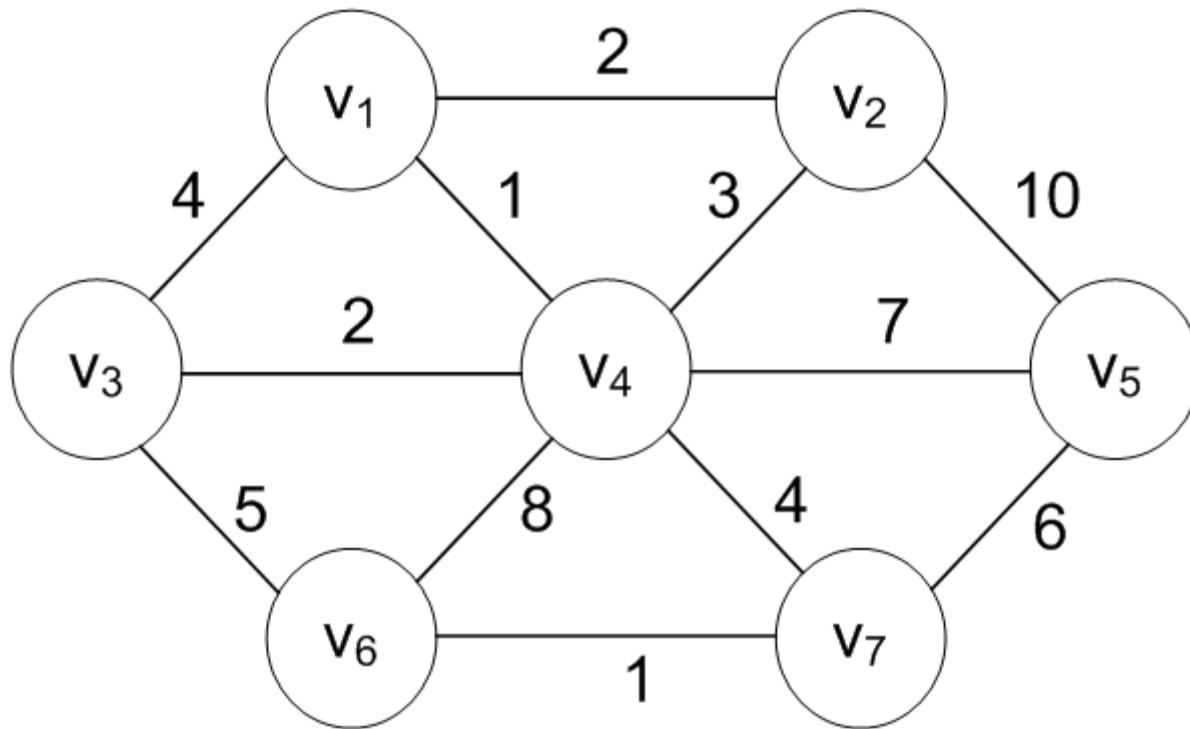
Une file de priorité est également utilisée



# Algorithme de Prim

---

Reprendons notre exemple:



# Algorithme de Prim

Nœuds	Distance	Connu?	Parent
V <sub>1</sub>	$\infty$	Faux	-
V <sub>2</sub>	$\infty$	Faux	-
V <sub>3</sub>	$\infty$	Faux	-
V <sub>4</sub>	$\infty$	Faux	-
V <sub>5</sub>	$\infty$	Faux	-
V <sub>6</sub>	$\infty$	Faux	-
V <sub>7</sub>	$\infty$	Faux	-

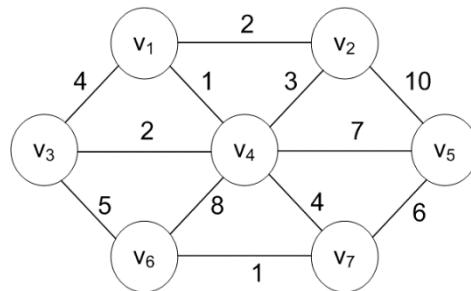


Nœuds	Distance	Connu?	Parent
V <sub>1</sub>	0	Vrai	-
V <sub>2</sub>	2	Faux	V <sub>1</sub>
V <sub>3</sub>	4	Faux	V <sub>1</sub>
V <sub>4</sub>	1	Faux	V <sub>1</sub>
V <sub>5</sub>	$\infty$	Faux	-
V <sub>6</sub>	$\infty$	Faux	-
V <sub>7</sub>	$\infty$	Faux	-

(V<sub>4</sub>, 1)  
(V<sub>2</sub>, 2)  
(V<sub>3</sub>, 4)

File de priorité

Entre (V<sub>1</sub>, 0)



File de priorité

Sort

(V<sub>1</sub>, 0)

Entrent

(V<sub>2</sub>, 2)

(V<sub>3</sub>, 4)

(V<sub>4</sub>, 1)

# Algorithme de Prim

Nœuds	Distance	Connu?	Parent	
$V_1$	0	Vrai	-	$(V_2, 2)$
$V_2$	2	Faux	$V_1$	$(V_3, 2)$
$V_3$	<u>2</u>	<u>Faux</u>	<u><math>V_4</math></u>	$(V_7, 4)$
$V_4$	1	<u>Vrai</u>	$V_1$	$(V_5, 7)$
$V_5$	7	Faux	$V_4$	$(V_6, 8)$
$V_6$	8	Faux	$V_4$	
$V_7$	4	Faux	$V_4$	



Nœuds	Distance	Connu?	Parent	
$V_1$	0	Vrai	-	$(V_3, 2)$
$V_2$	2	Vrai	$V_1$	$(V_7, 4)$
$V_3$	2	Faux	$V_4$	$(V_5, 7)$
$V_4$	1	Vrai	$V_1$	$(V_6, 8)$
$V_5$	7	Faux	$V_4$	
$V_6$	8	Faux	$V_4$	
$V_7$	4	Faux	$V_4$	

## File de priorité

Sort

$(V_4, 1)$

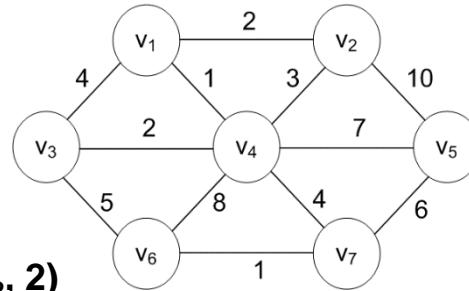
Entrent

$(V_5, 7)$

$(V_6, 8)$

$(V_7, 4)$

Inchangé       $(V_2, 2)$   
Change       $(V_3, 2)$



## File de priorité

Sort

$(V_2, 2)$

Inchangé       $(V_5, 7)$

# Algorithme de Prim

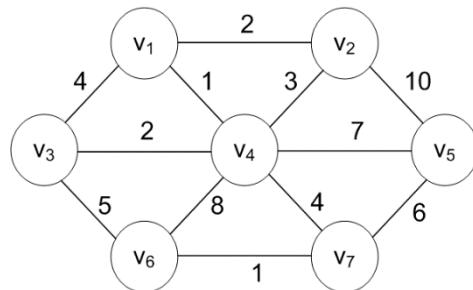
Nœuds	Distance	Connu?	Parent		Nœuds	Distance	Connu?	Parent	
V <sub>1</sub>	0	Vrai	-	(V <sub>7</sub> , 4)	V <sub>1</sub>	0	Vrai	-	(V <sub>6</sub> , 1)
V <sub>2</sub>	2	Vrai	V <sub>1</sub>	(V <sub>6</sub> , 5)	V <sub>2</sub>	2	Vrai	V <sub>1</sub>	(V <sub>5</sub> , 6)
V <sub>3</sub>	2	Vrai	V <sub>4</sub>	(V <sub>5</sub> , 7)	V <sub>3</sub>	2	Vrai	V <sub>4</sub>	
V <sub>4</sub>	1	Vrai	V <sub>1</sub>	—————>	V <sub>4</sub>	1	Vrai	V <sub>1</sub>	
V <sub>5</sub>	7	Faux	V <sub>4</sub>		V <sub>5</sub>	6	Faux	V <sub>7</sub>	
<b>V<sub>6</sub></b>	<b>5</b>	<b>Faux</b>	<b>V<sub>3</sub></b>		<b>V<sub>6</sub></b>	<b>1</b>	<b>Faux</b>	<b>V<sub>7</sub></b>	
V <sub>7</sub>	4	Faux	V <sub>4</sub>		V <sub>7</sub>	4	Vrai	V <sub>4</sub>	

File de priorité

Sort

(V<sub>3</sub>, 3)

Change (V<sub>6</sub>, 5)



File de priorité

Sort

(V<sub>7</sub>, 4)

Change (V<sub>6</sub>, 1)  
(V<sub>5</sub>, 6)

# Algorithme de Prim

( $V_5, 6$ )

Nœuds	Distance	Connu?	Parent
$V_1$	0	Vrai	-
$V_2$	2	Vrai	$V_1$
$V_3$	2	Vrai	$V_4$
$V_4$	1	Vrai	$V_1$
$V_5$	6	Faux	$V_7$
$V_6$	1	Vrai	$V_7$
$V_7$	4	Vrai	$V_4$

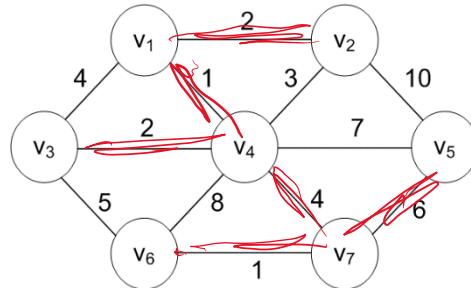


Nœuds	Distance	Connu?	Parent
$V_1$	0	Vrai	-
$V_2$	2	Vrai	$V_1$
$V_3$	2	Vrai	$V_4$
$V_4$	1	Vrai	$V_1$
$V_5$	6	Vrai	$V_7$
$V_6$	1	Vrai	$V_7$
$V_7$	4	Vrai	$V_4$

File de priorité

Sort

( $V_6, 1$ )



File de priorité

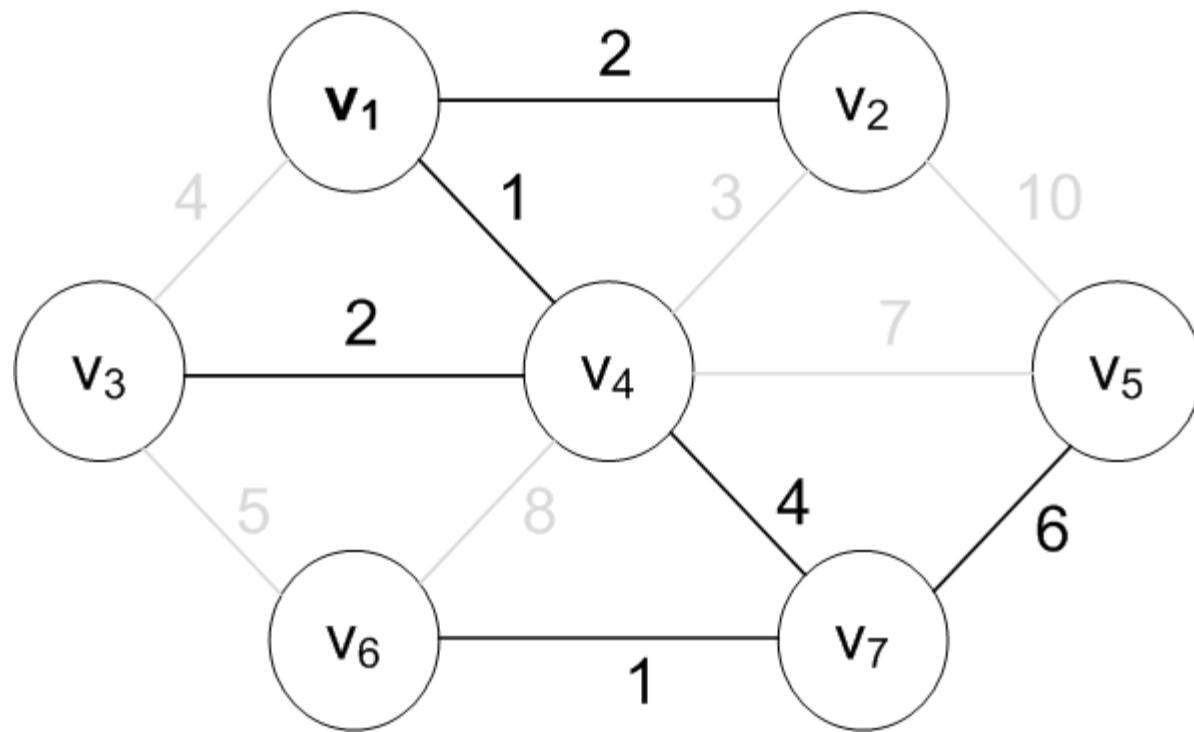
Sort

( $V_5, 6$ )

# Algorithme de Prim

---

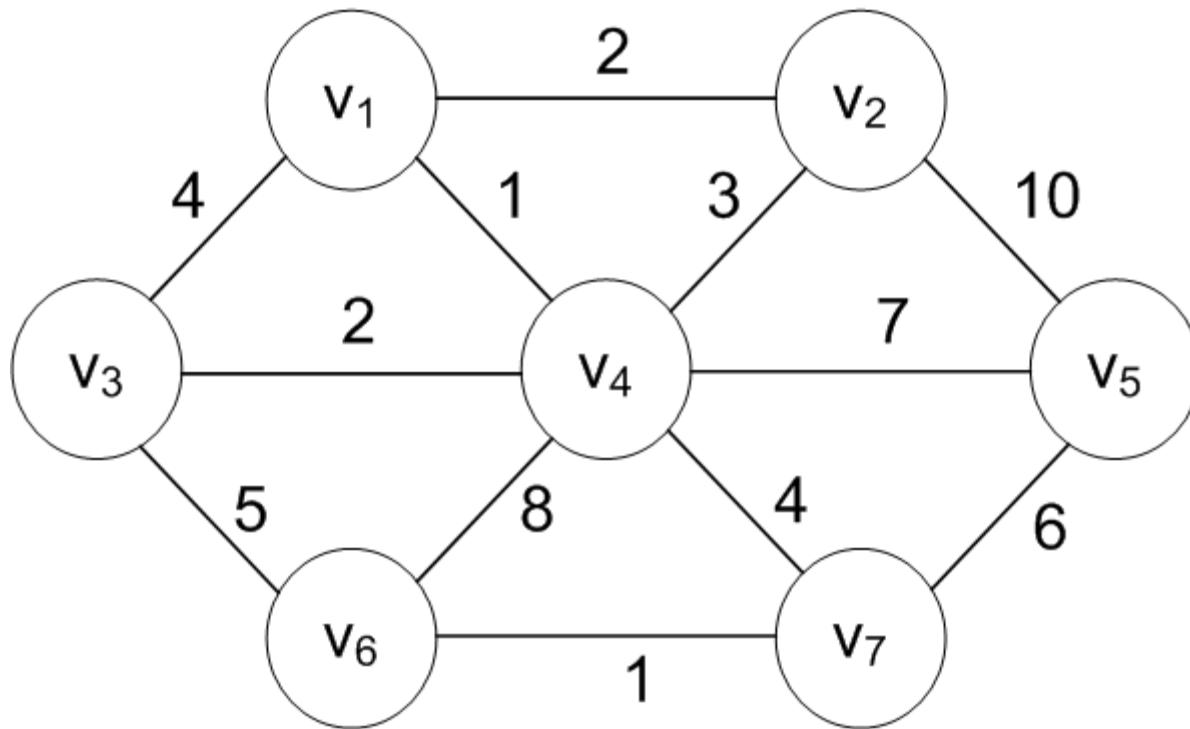
Et on parvient à la solution:



# Algorithme de Prim

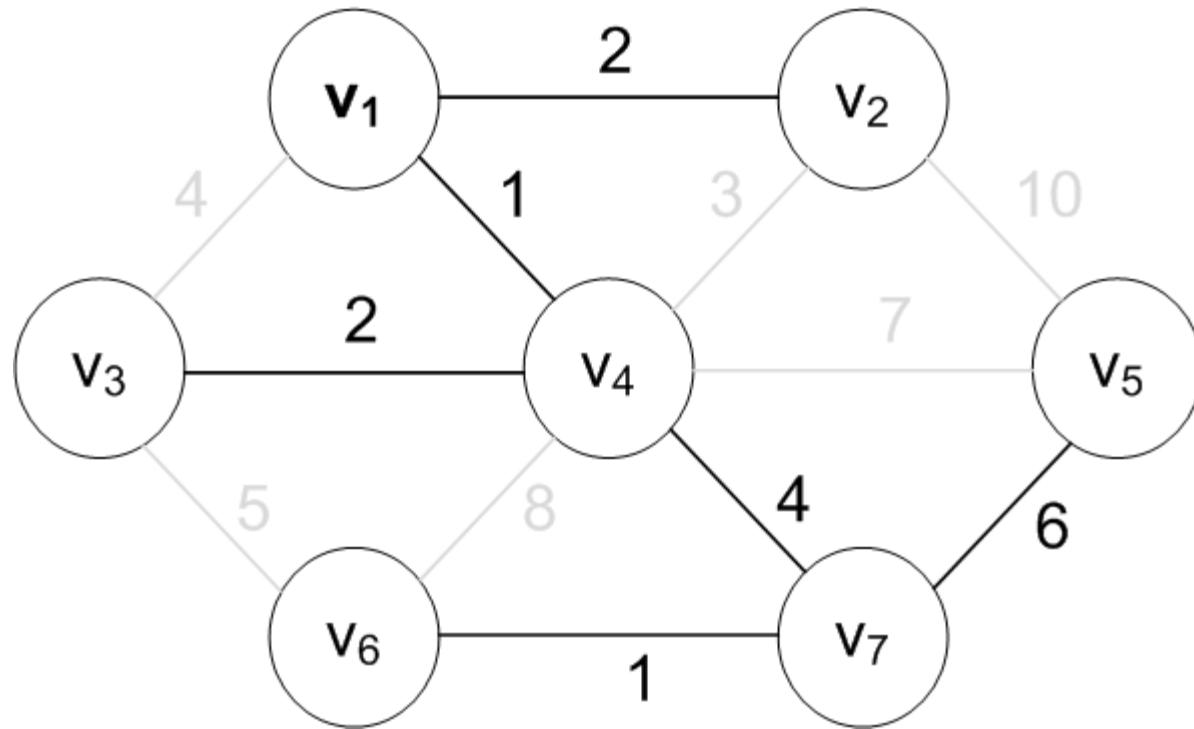
---

Quel résultat aurions-nous si on partait de  $v_5$  ?



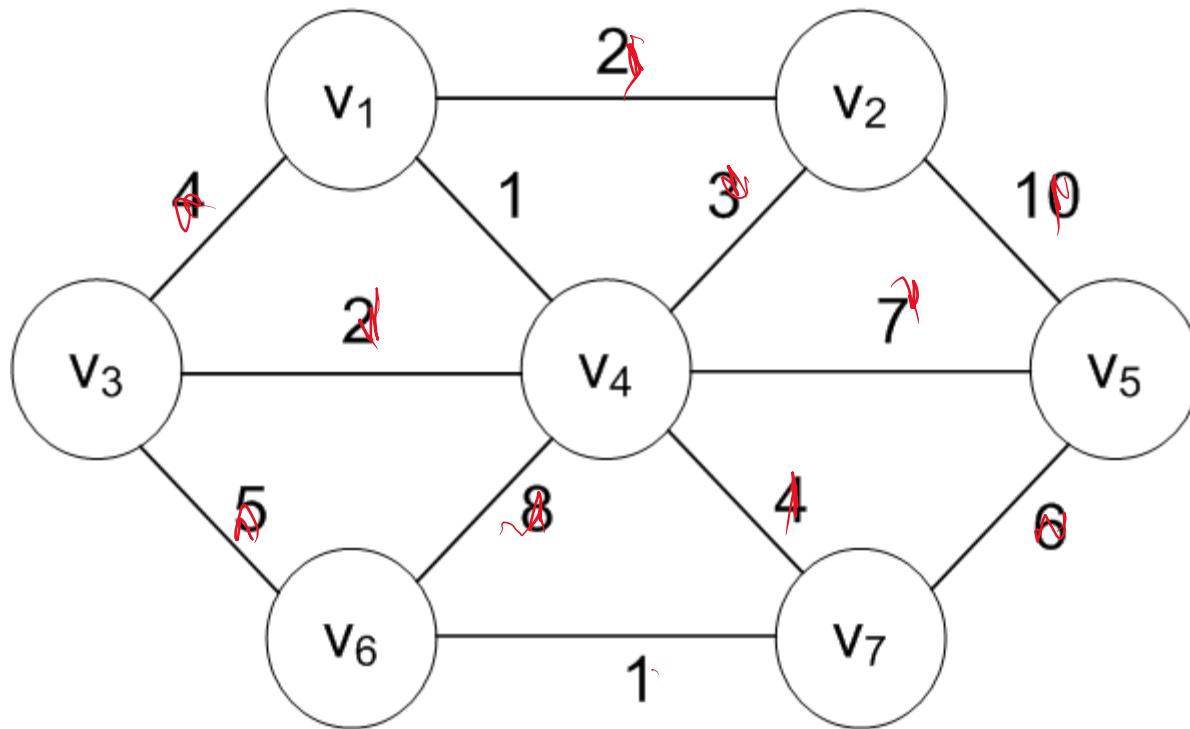
# Algorithme de Prim

Et on parvient à la même solution:



# Algorithme de Prim

Quel résultat aurions-nous si on partait de  $v_5$  ?



# Graphes II

---

1. Implementation
    1. Interface et classes de graphes orientés et non orientés
    2. Class Paths (BFS + DFS)
  2. Ordre topologique (version DFS)
    1. Parcours DFS post-ordre et post-ordre inverse
    2. Algorithme d'ordre topologique
  3. Composantes connexes
    1. Notion de connexité
    2. Composantes connexes (UG)
    3. Composantes fortement connexes (DG)
  4. Arbre sous-tendant minimum
    1. Problématique
    2. Algorithme de Prim
    3. Algorithme de Kruskal
-

# Algorithme de Kruskal

---

## Définition

L'algorithme de Kruskal permet de trouver l'arbre sous-tendant minimum d'un graphe et s'exprime comme suit:

1. Créer une forêt **F** à partir des sommets du graphe
2. Créer une collection **A** contenant tous les arcs du graphe

Tant que **A** n'est pas vide et que **F** contient plus d'un arbre

    Retirer de **A** l'arc le plus petit poids

    Si l'arc lie deux arbres différents dans **F**,

        Union des deux arbres

# Algorithme de Kruskal

## Définition

L'algorithme de Kruskal permet de trouver l'arbre sous-tendant minimum d'un graphe et s'exprime comme suit:

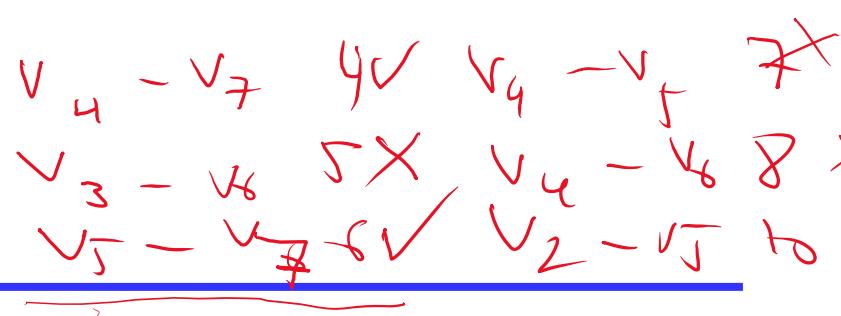
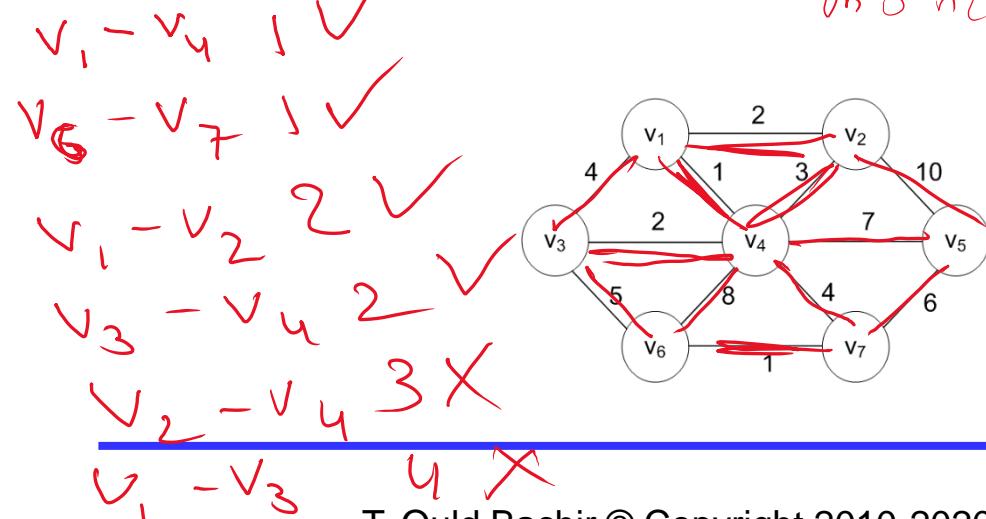
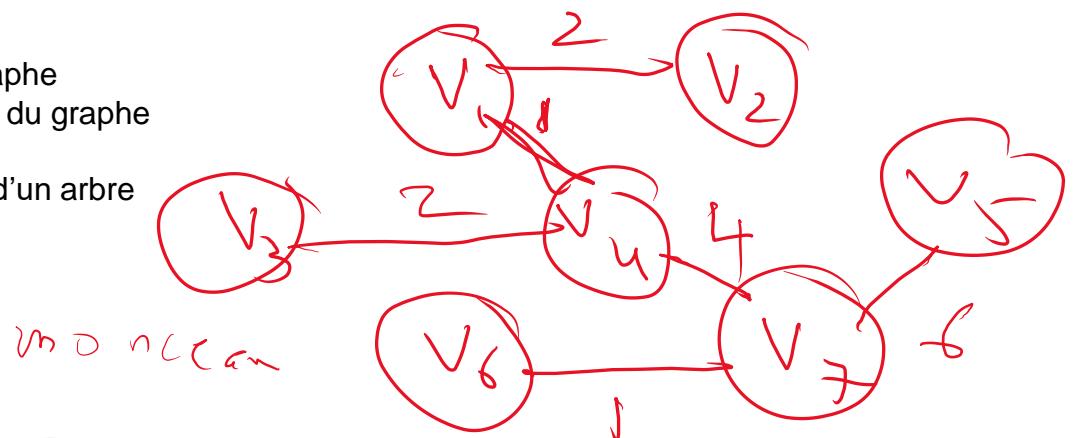
1. Créer une forêt **F** à partir des sommets du graphe
2. Créer une collection **A** contenant tous les arcs du graphe

Tant que **A** n'est pas vide et que **F** contient plus d'un arbre

Retirer de **A** l'arc le plus petit poids

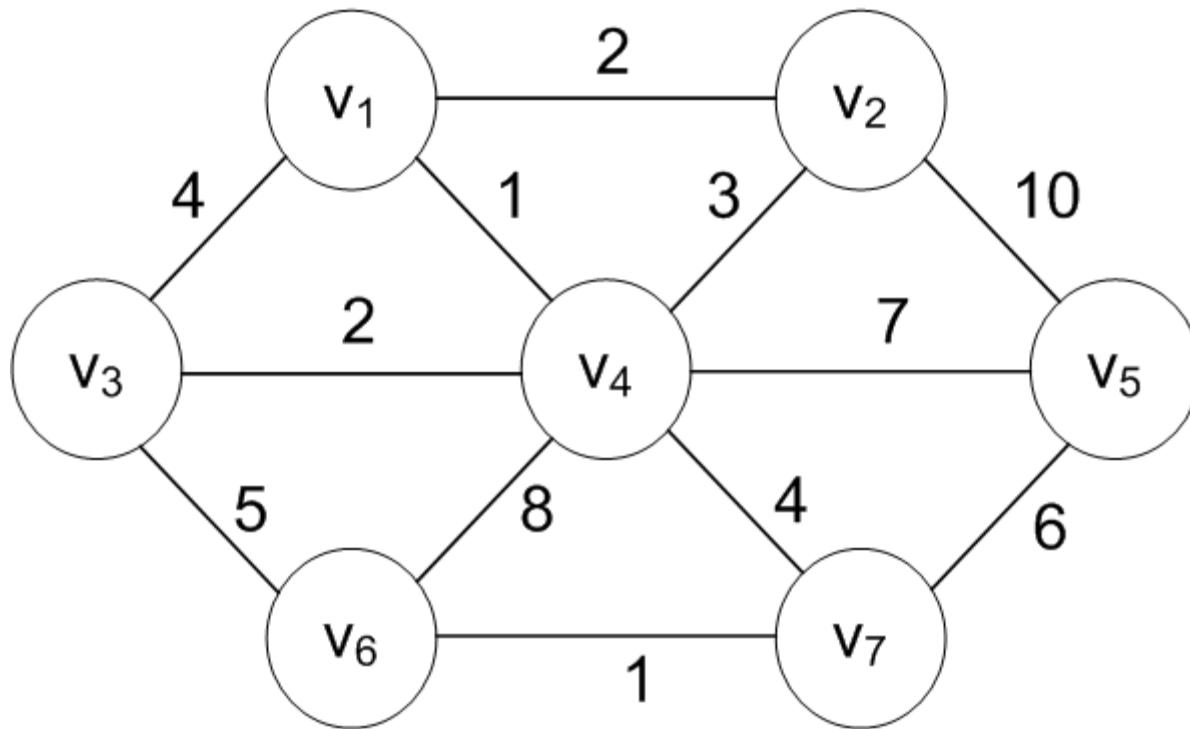
Si l'arc lie deux arbres différents dans **F**,

Union des deux arbres



# Algorithme de Kruskal

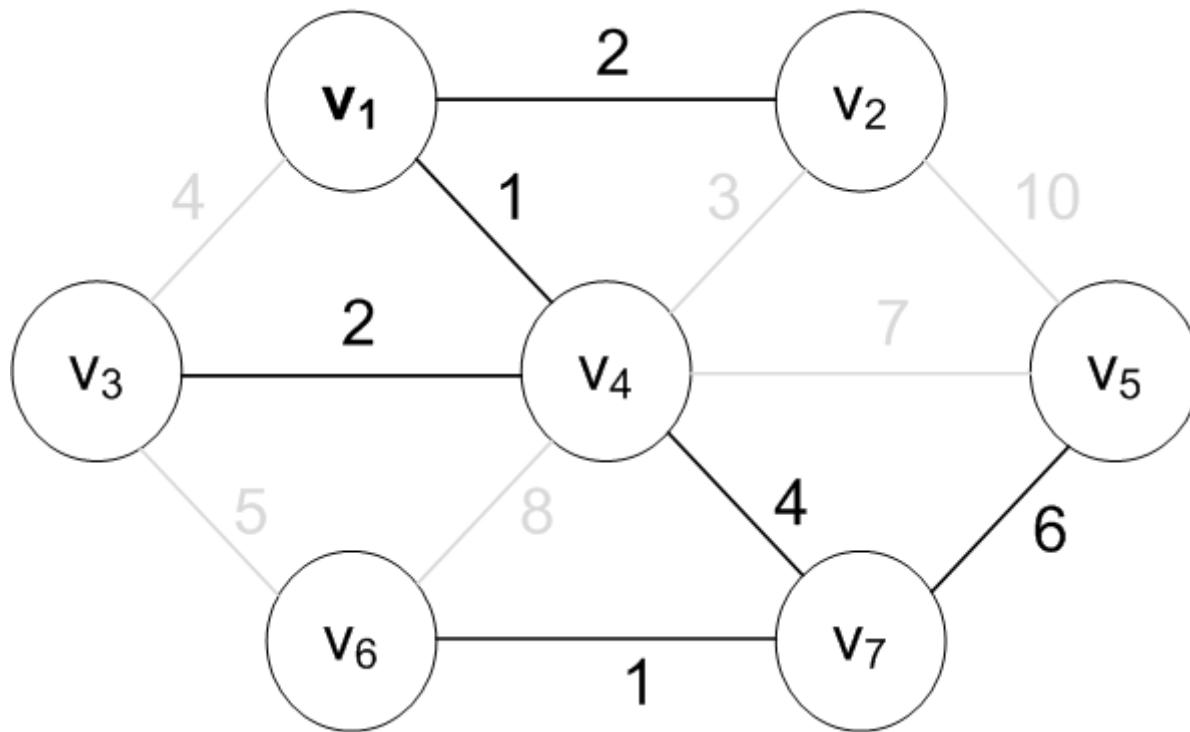
Considérons le graphe valué et non dirigé suivant, pour lequel on cherche l'arbre sous-tendant minimum:



# Algorithme de Kruskal

---

On cherche à obtenir ceci:



---

# **INF2010 – ASD**

## Algorithmes sur les chaînes de caractères

# Plan

---

Recherche de patron

Problématique

Rabin Karp

Automate FSM

PLSC

Problématique

Solution par programmation dynamique

# Plan

---

Recherche de patron

Problématique

Rabin Karp

Automate FSM

PLSC

Problématique

Solution par programmation dynamique

# Problématique

---

## Problématique:

Chercher la chaîne de caractères  $P[1..m]$  dans un texte  $T[1..n]$ .  
où  $m \leq n$ .

Ayant  $m \leq n$ , on traduit le problème par chercher tous les  
 $s \leq n-m+1$  pour lesquels  $T[s+1...s+m] = P[1..m]$

## Exemples:

- Chercher un mot dans un fichier
  - Chercher un fichier dans un volume (HDD, Clé Flash, CD-ROM)
  - Chercher un mot dans des fichiers
  - Chercher un mot dans le web (google, yahoo)
-

# Algorithme naïf

---

**Pour**  $s=0$  à  $n-m$

**Pour**  $j= 1$  à  $m$

**Si**  $T[s+j] \neq P[j]$

**Reprendre** au  $s$  suivant

**Sinon Si**  $j=m$

**Inclure**  $s$  dans  $S$

**Retourner**  $S$

# Algorithme naïf

---

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

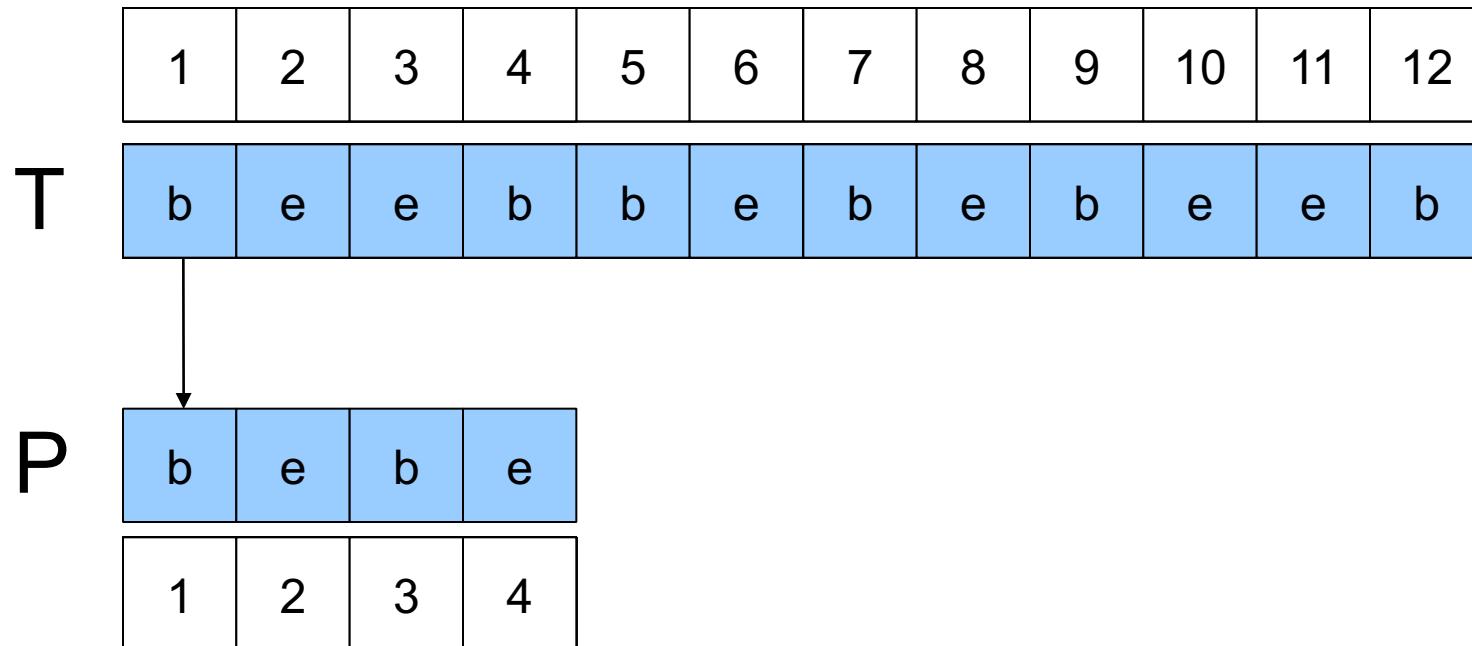
T	b	e	e	b	b	e	b	e	b	e	e	b
---	---	---	---	---	---	---	---	---	---	---	---	---

P	b	e	b	e
	1	2	3	4

# Algorithme naïf

---

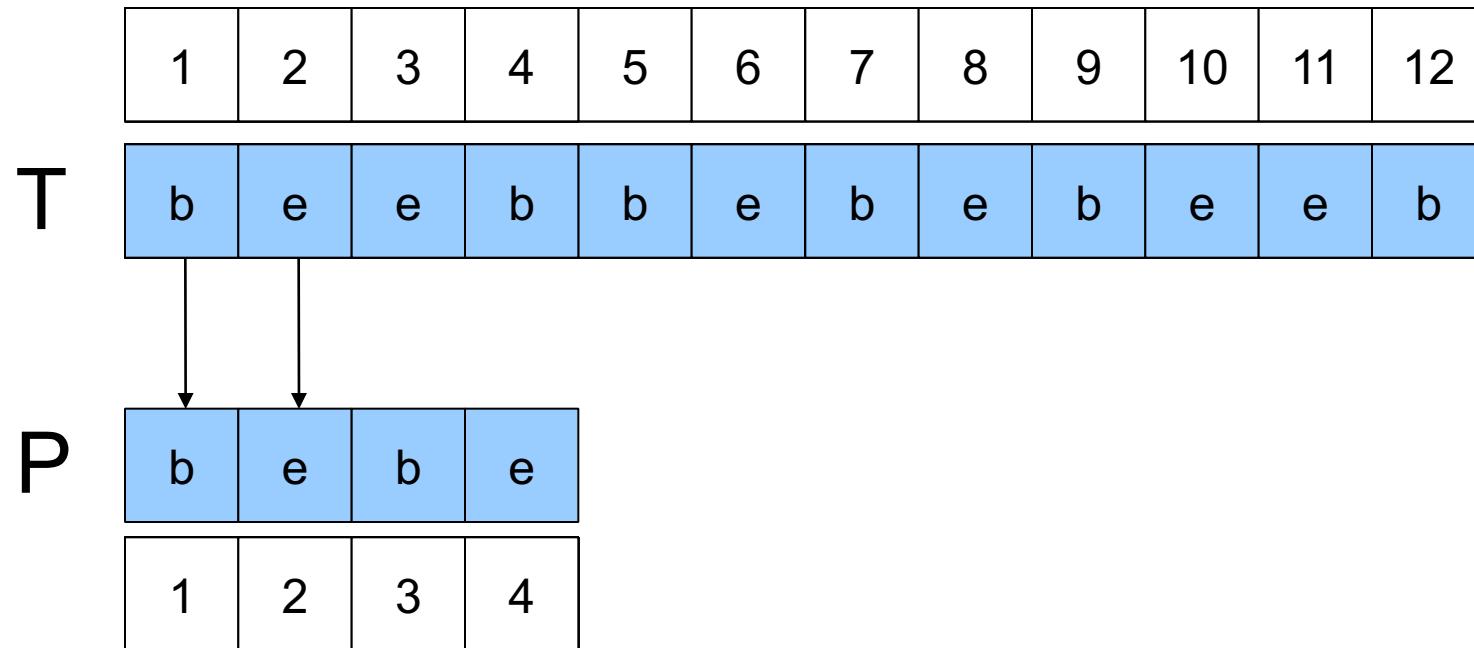
$s=0, j=1$



# Algorithme naïf

---

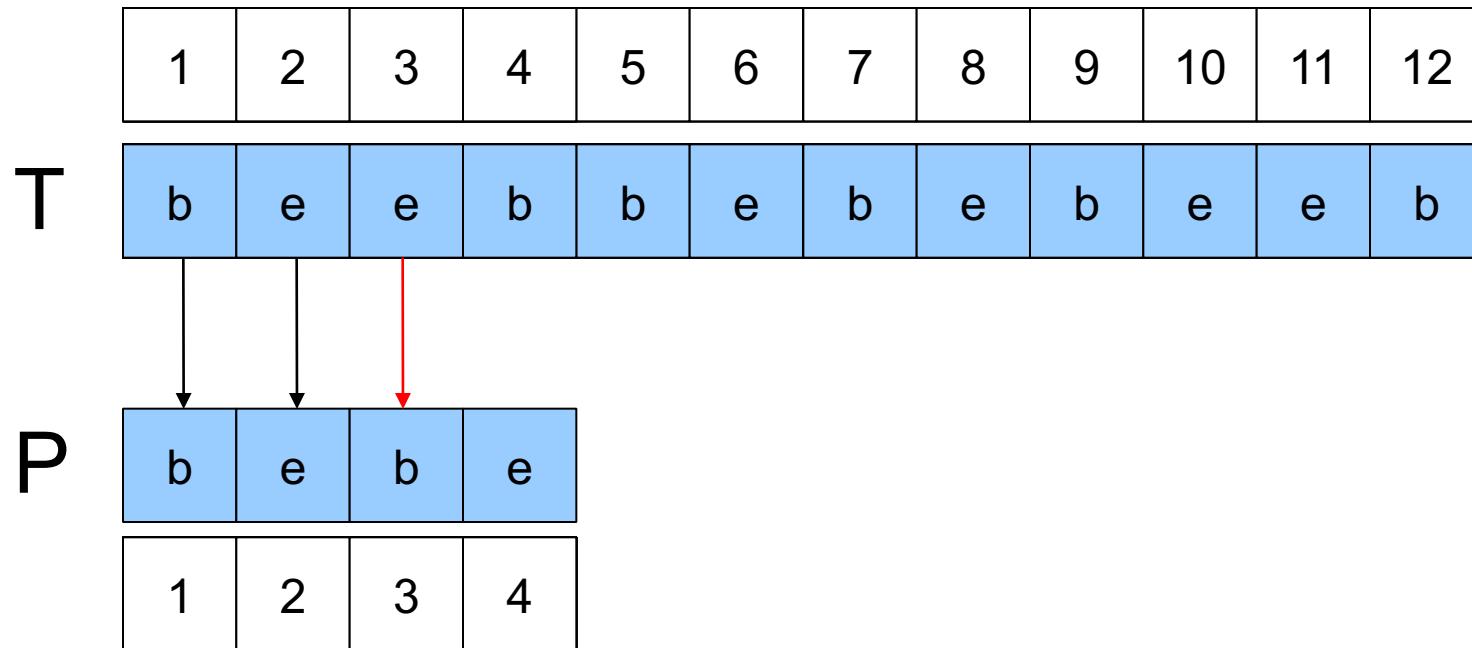
$s=0, j=2$



# Algorithme naïf

---

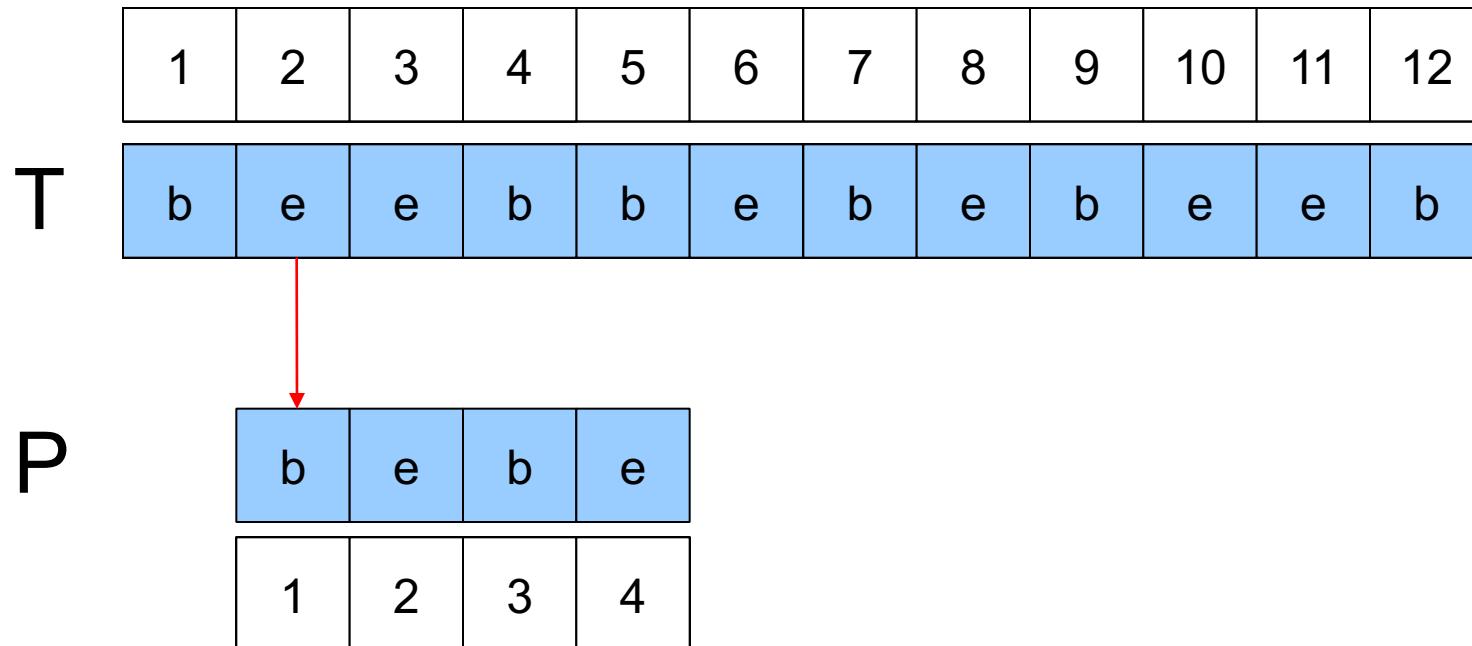
$s=0, j=3$



# Algorithme naïf

---

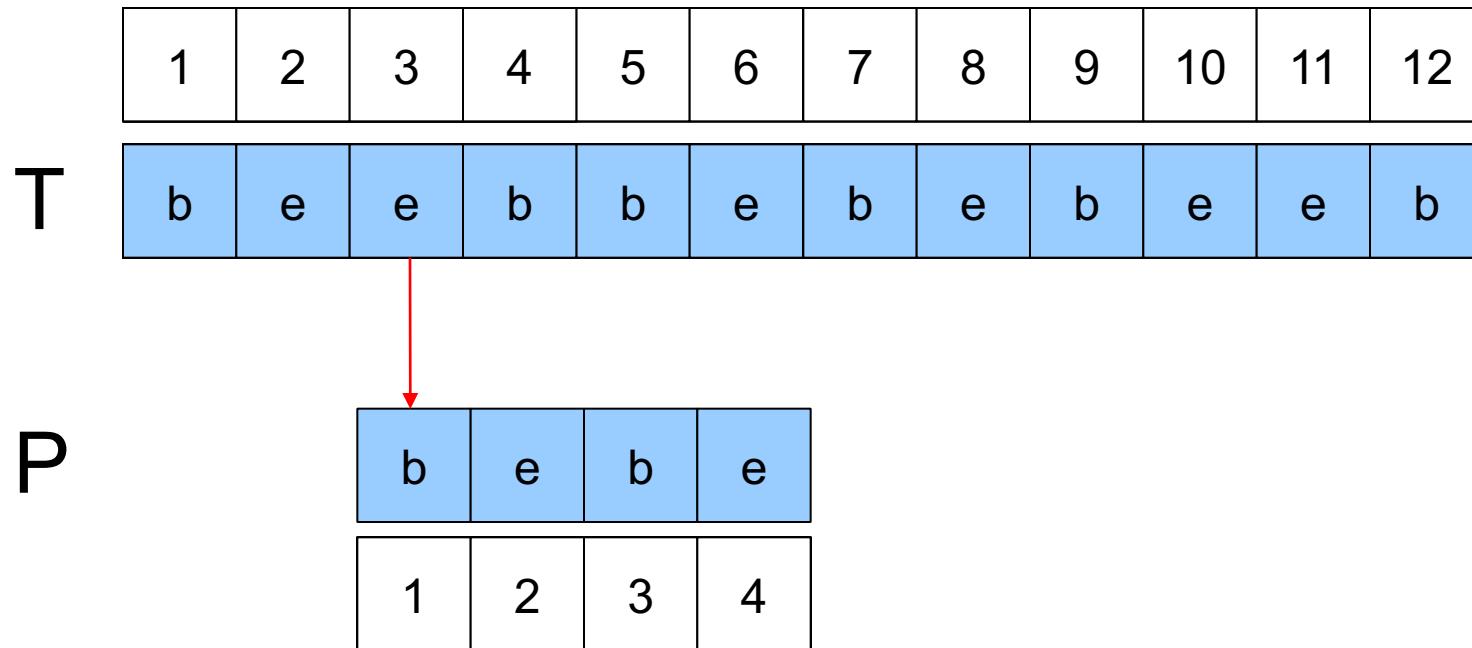
$s=1, j=1$



# Algorithme naïf

---

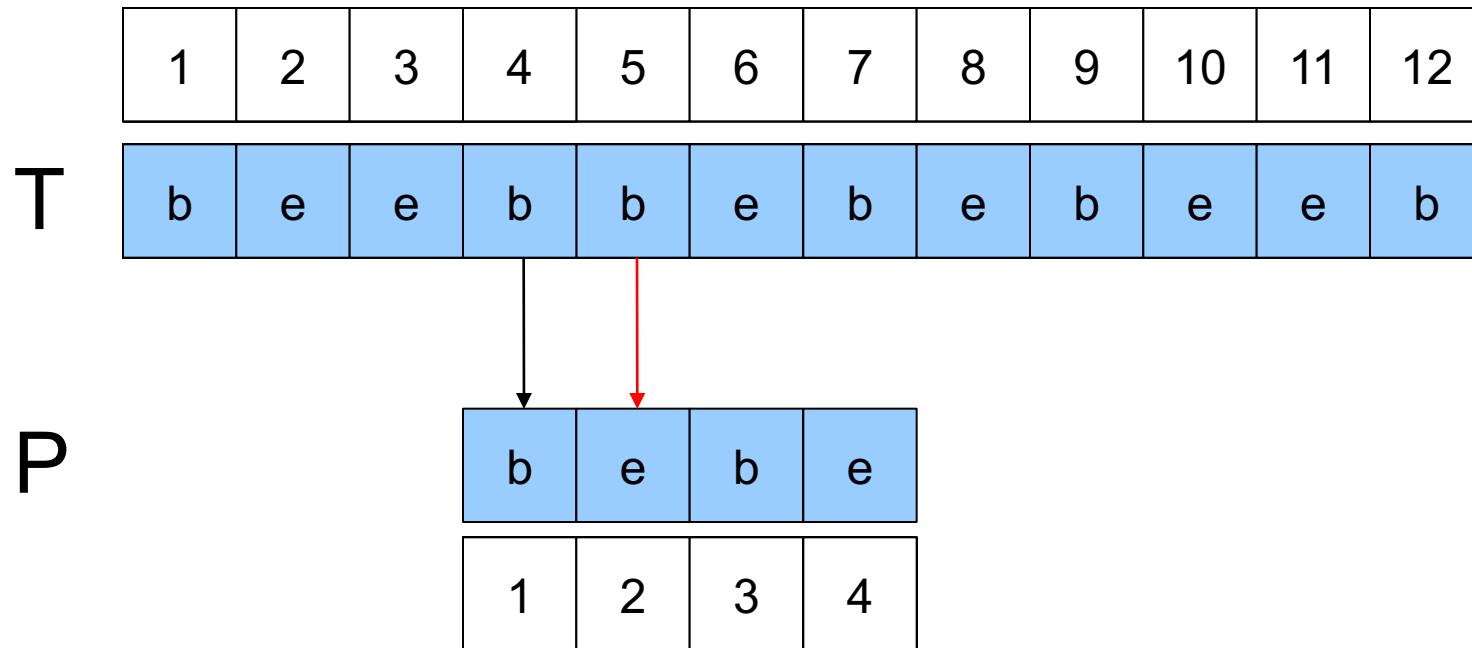
$s=2, j=1$



# Algorithme naïf

---

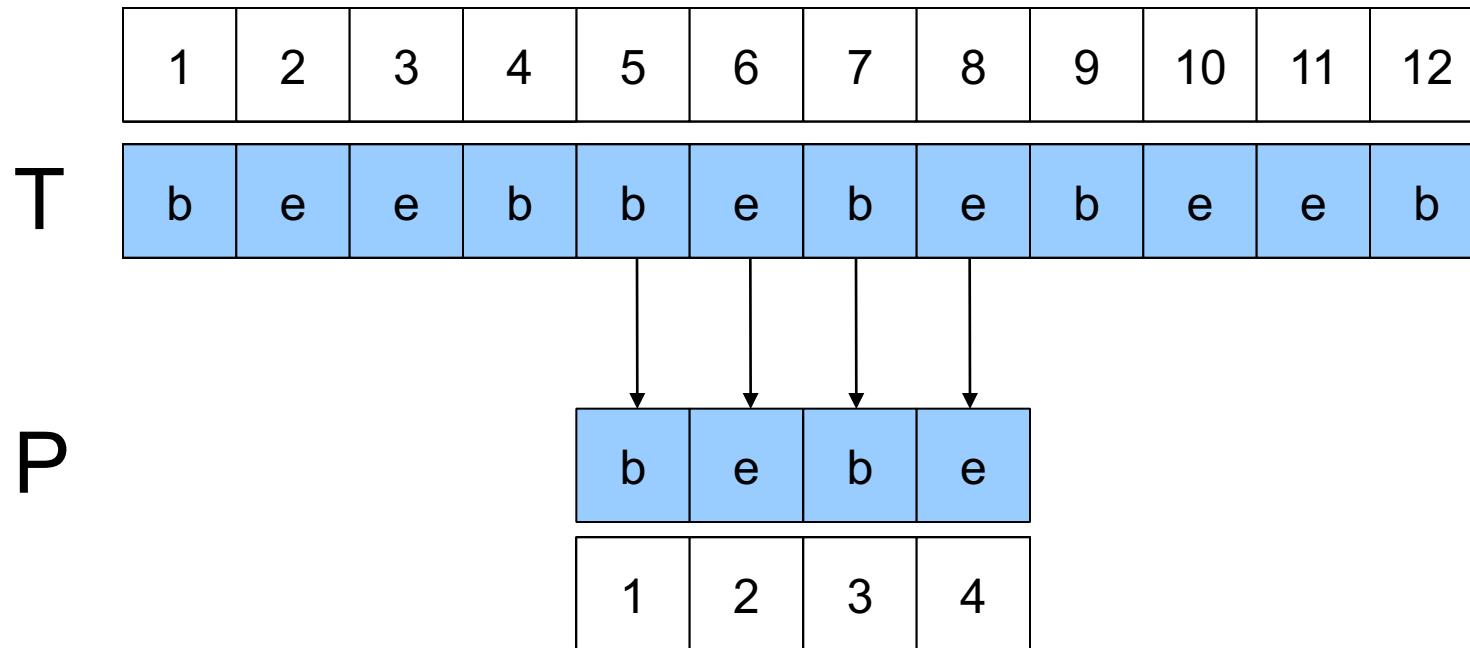
$s=3, j=2$



# Algorithme naïf

$s=4, j=4=m$

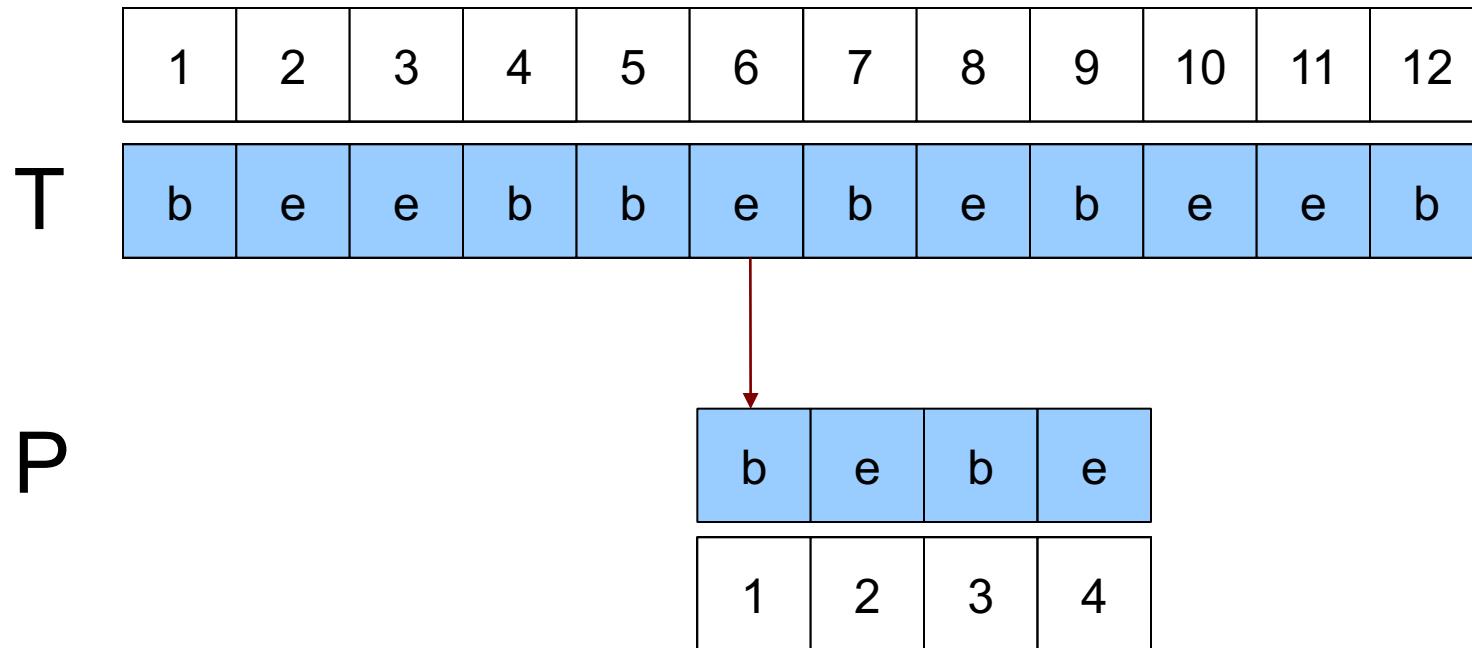
$S=\{4\}$



# Algorithme naïf

$s=5, j=1$

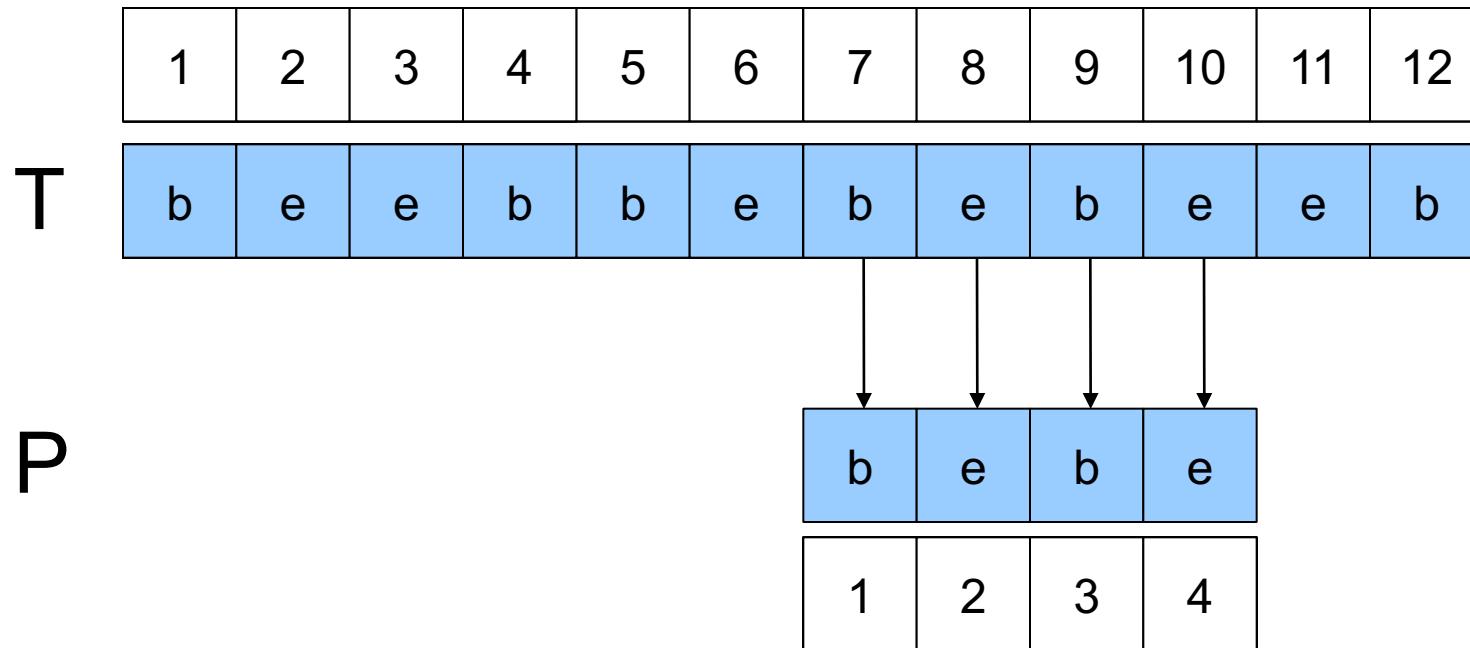
$S=\{4\}$



# Algorithme naïf

$s=6, j=4$

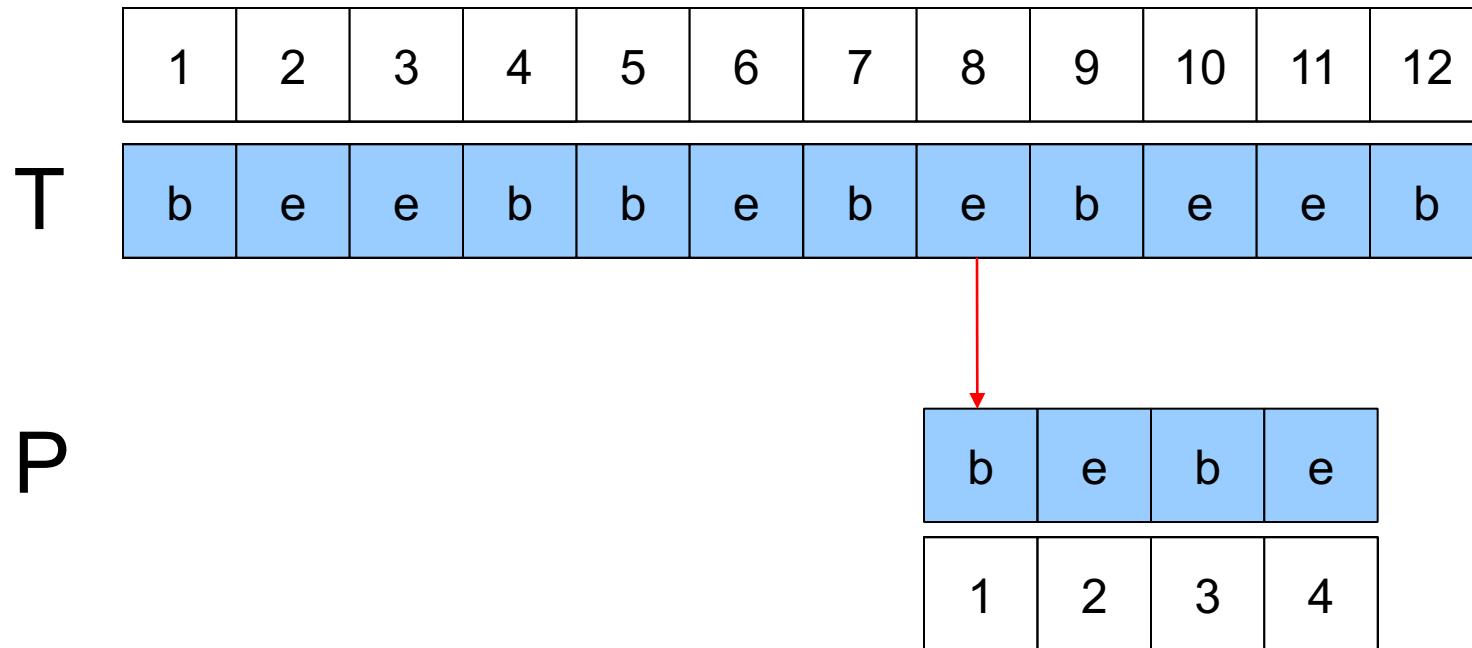
$S=\{4,6\}$



# Algorithme naïf

$s=7, j=1$

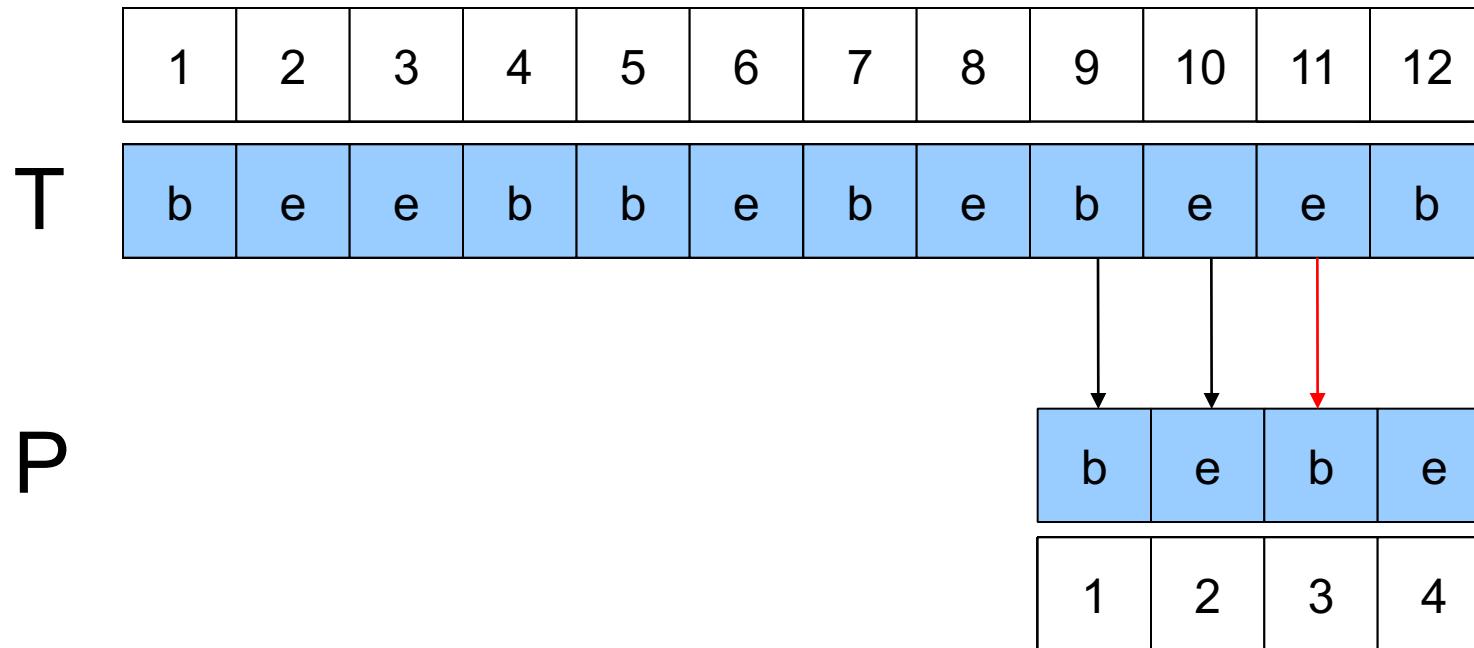
$S=\{4,6\}$



# Algorithme naïf

$$s=8=12-4=n-m, j=3$$

$$S=\{4,6\}$$

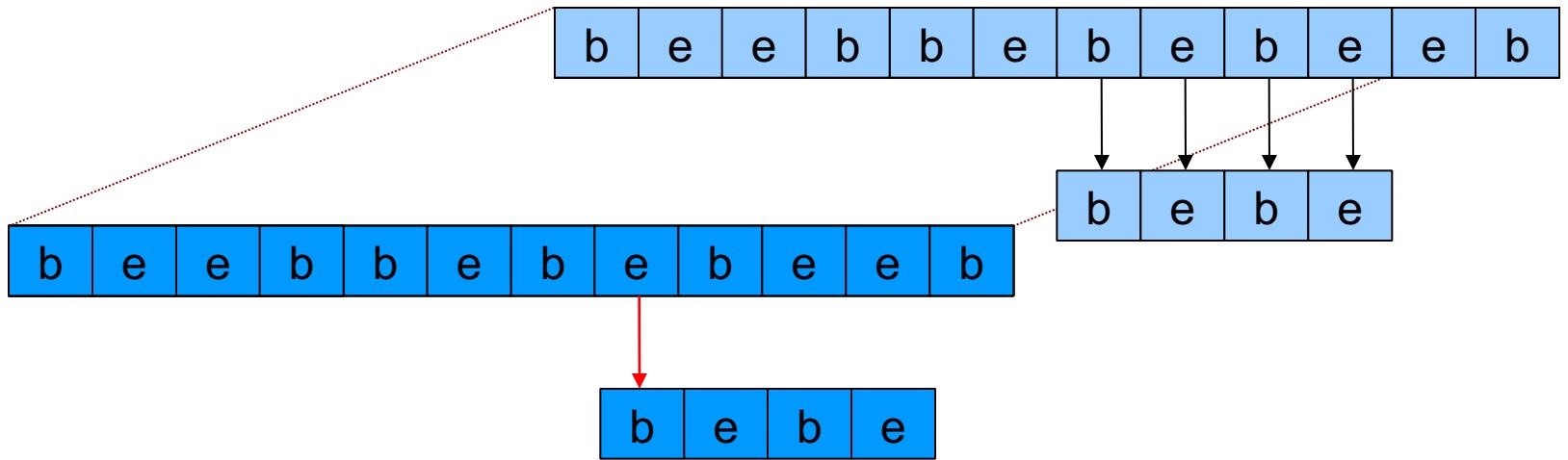


# Algorithme naïf

Complexité de l'algorithme?

$$O(m(n-m+1))$$

Idée intéressante: réutiliser les résultats et analyser la chaîne de caractère P



# Plan

---

Recherche de patron

Problématique

Rabin Karp

Automate FSM

PLSC

Problématique

Solution par programmation dynamique

# Rabin-Karp

---

Objectif: battre l'algorithme naïf de complexité

$$O(m(n-m+1))$$

Analyser la chaîne de caractère:  
prétraitements (*preprocessing*)

Réutiliser les résultats précédents:  
accélération

# Rabin-Karp: formulation 1

---

Idée:

Pour un alphabet  $\Sigma$ , écrire  $P$  sous forme d'un nombre  $p$  dans la base  $d=|\Sigma|$

Analyser la chaîne de caractère:

Trouver une fois la valeur  $p$  associée à  $P$

# Rabin-Karp: formulation 1

---

Pour l'exemple, on utilise les chiffre 0-9:

Pour un alphabet  $\Sigma=\{0, 1, 2, \dots, 8, 9\}$ , écrire  $P$  sous forme d'un nombre dans la base  $d=|\Sigma|=10$

Exemple:

$P$ 

1	2	3	5
---	---	---	---

  
 $p = 1\ 235$

# Rabin-Karp: formulation 1

---

P    

1	2	3	5
---	---	---	---

$$p = 1\ 235$$

$$p = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$$

Pour une longueur quelconque  $m$ , la phase de prétraitement peut s'avérer ardue.

# Rabin-Karp: formulation 1

---

P    

1	2	3	5
---	---	---	---

$$p = 1\ 235$$

$$p = ((1 \cdot 10^1 + 2) \cdot 10^1 + 3) \cdot 10^1 + 5 \cdot 10^0$$

En utilisant l'algorithme de Horner, on réduit le nombre d'opérations pour le prétraitement.

# Rabin-Karp: formulation 1

---

**Calculer** par Horner  $p$  de  $P[1..m]$

**Pour**  $s=0$  à  $n-m$

**Calculer** par Horner  $t_s$  de  $T[s+1..s+m]$

**Si**  $t_s = p$

**Inclure**  $s$  dans  $S$

**Retourner**  $S$

# Rabin-Karp: formulation 1

---

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

T	1	3	2	3	4	1	2	3	5	1	2	3
---	---	---	---	---	---	---	---	---	---	---	---	---

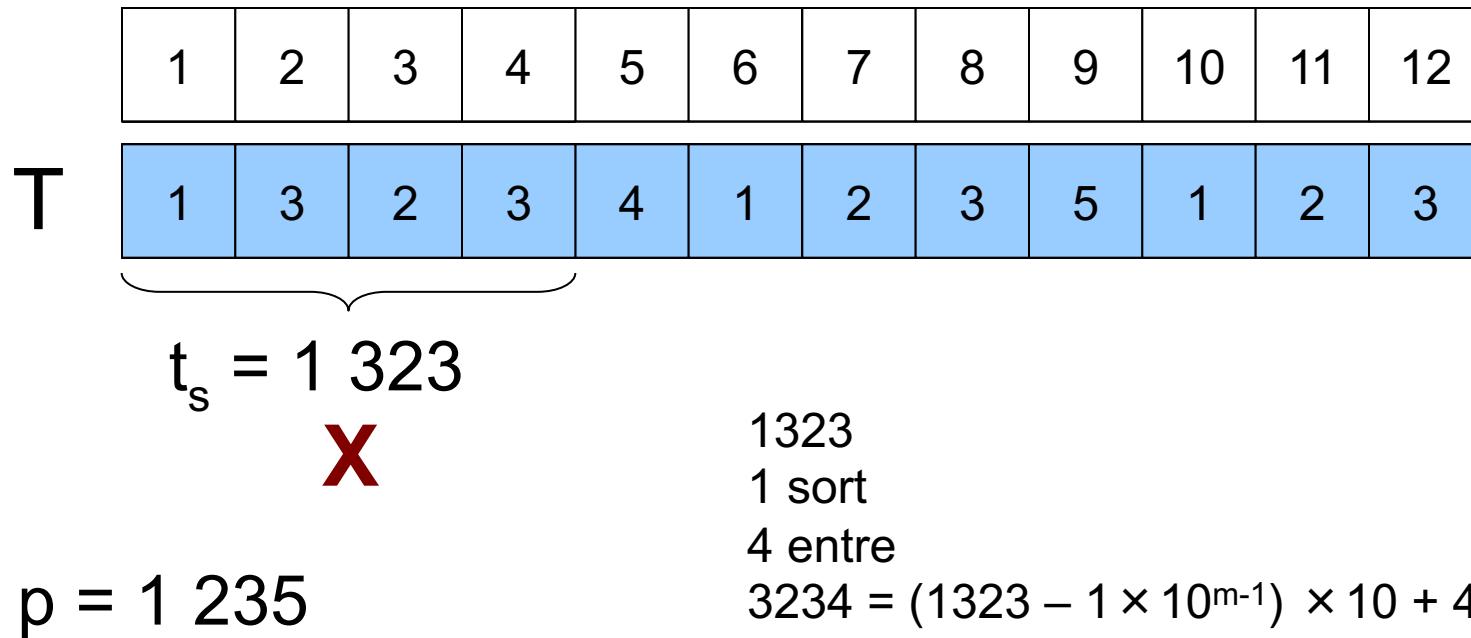
P	1	2	3	5
---	---	---	---	---

$p = 1\ 235$

# Rabin-Karp: formulation 1

---

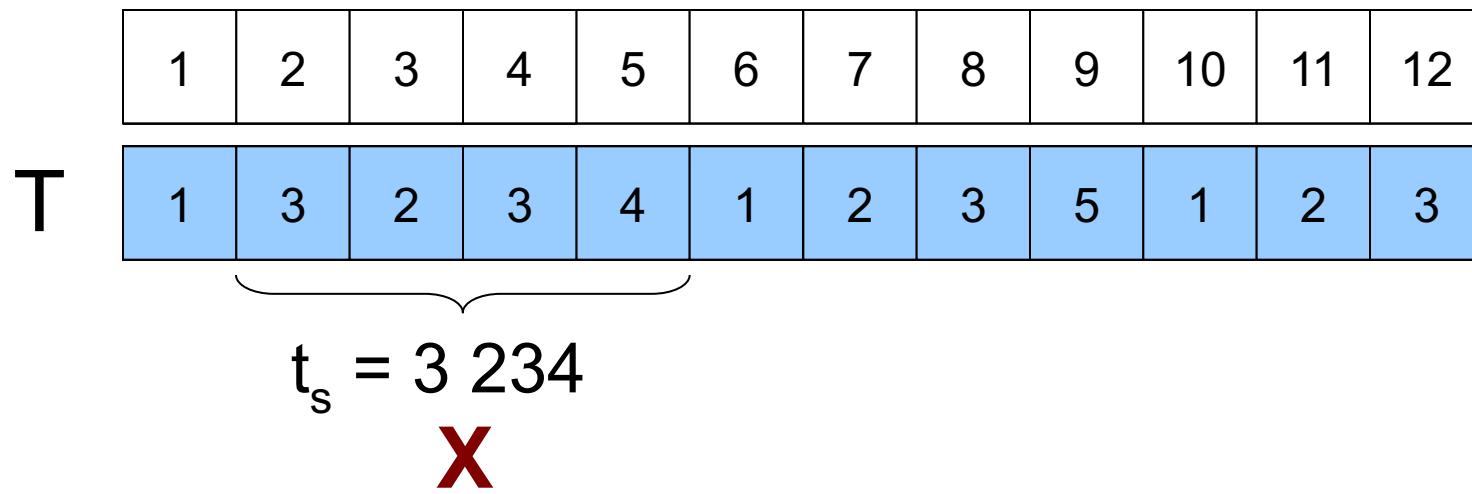
$s=0$



# Rabin-Karp: formulation 1

---

$s=1$



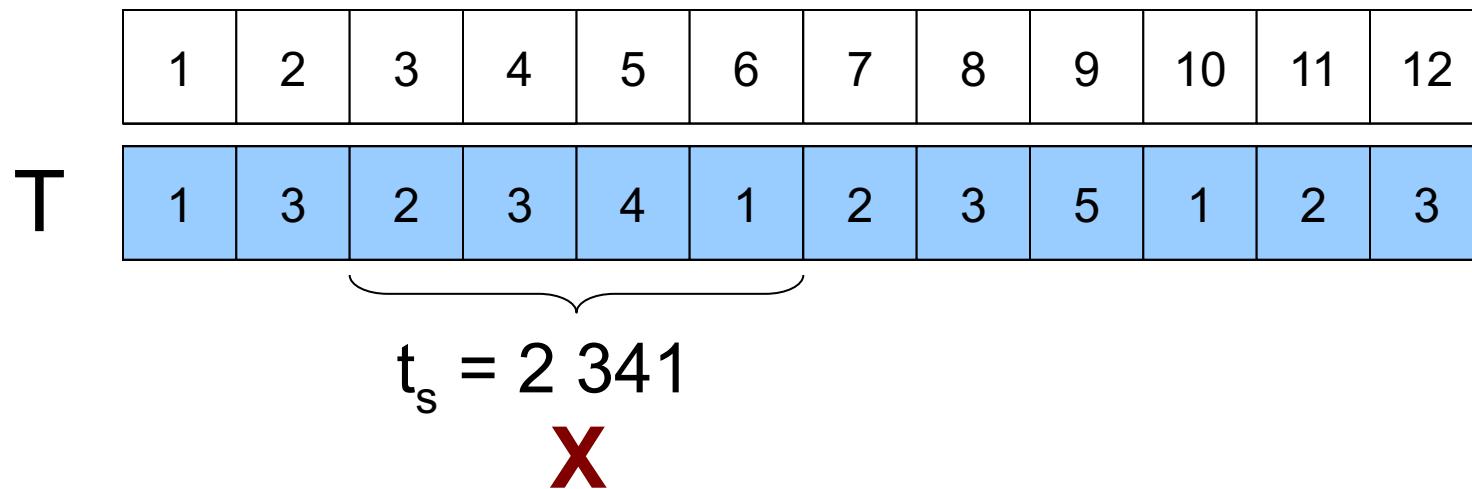
$p = 1\ 235$

---

# Rabin-Karp: formulation 1

---

$s=2$



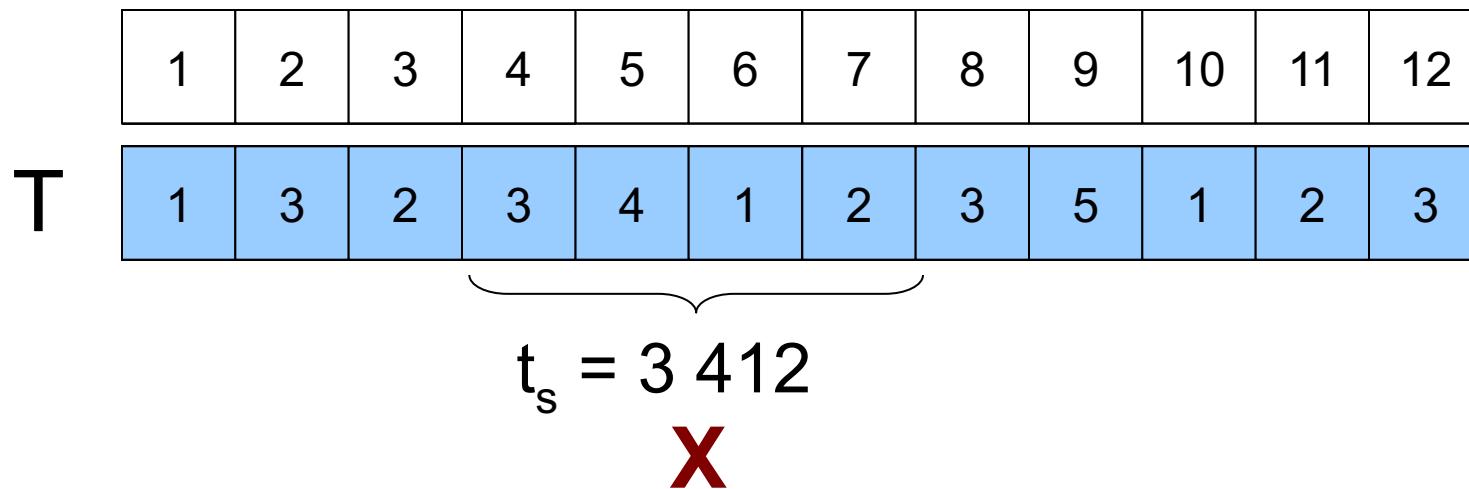
$p = 1\ 235$

---

# Rabin-Karp: formulation 1

---

$s=3$



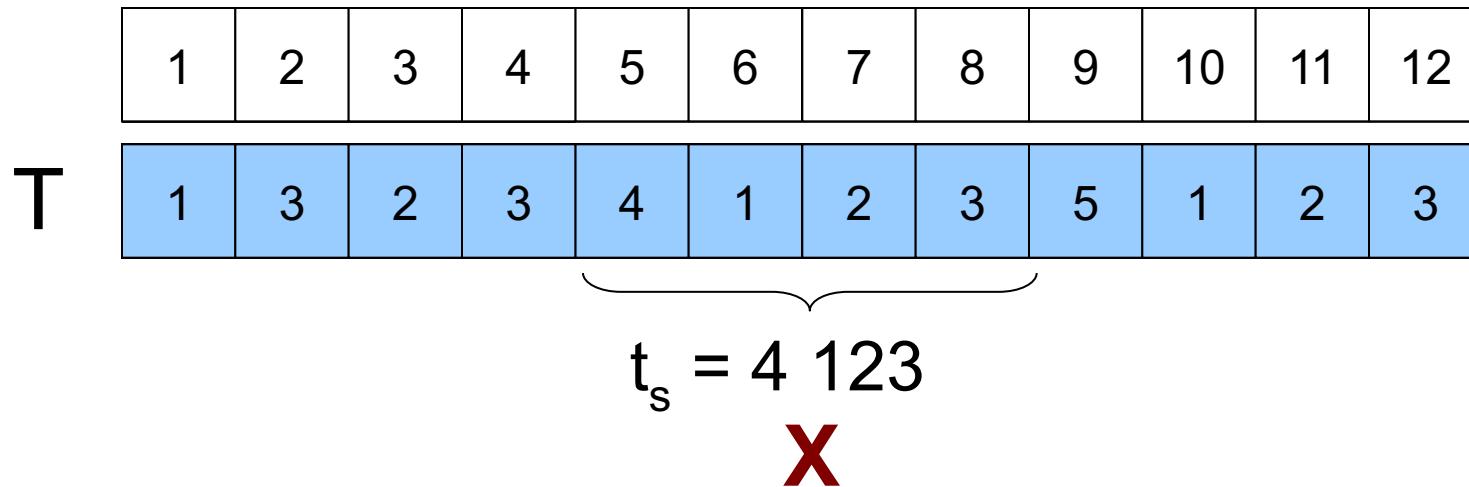
$p = 1\ 2\ 3\ 5$

---

# Rabin-Karp: formulation 1

---

s=4



$p = 1\ 235$

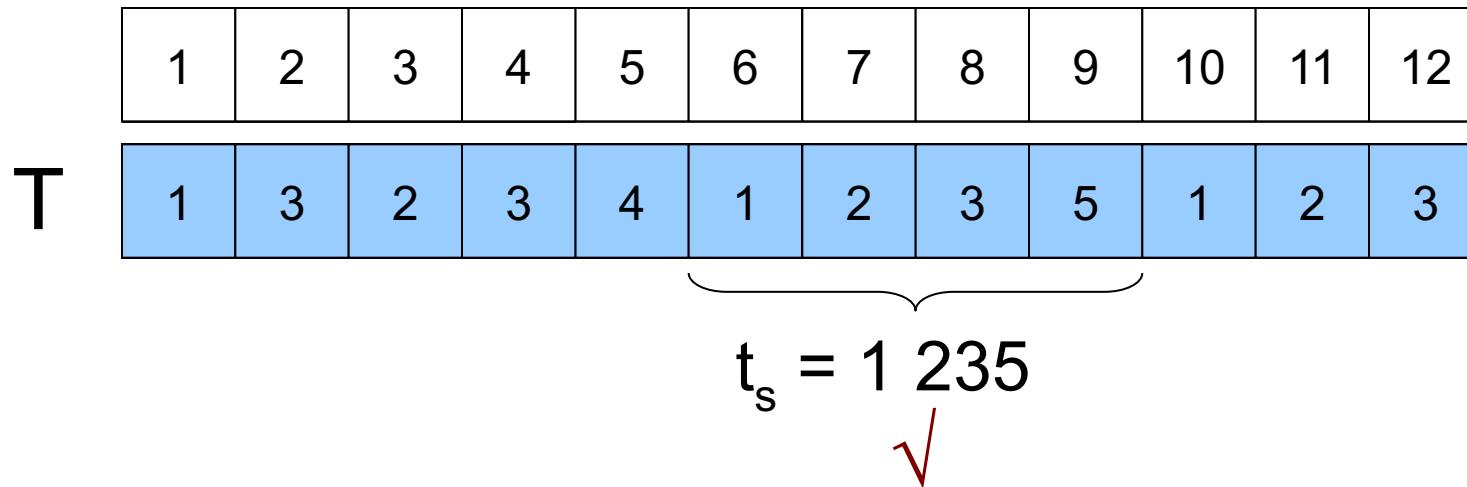
---

# Rabin-Karp: formulation 1

---

$s=5$

$S=\{5\}$



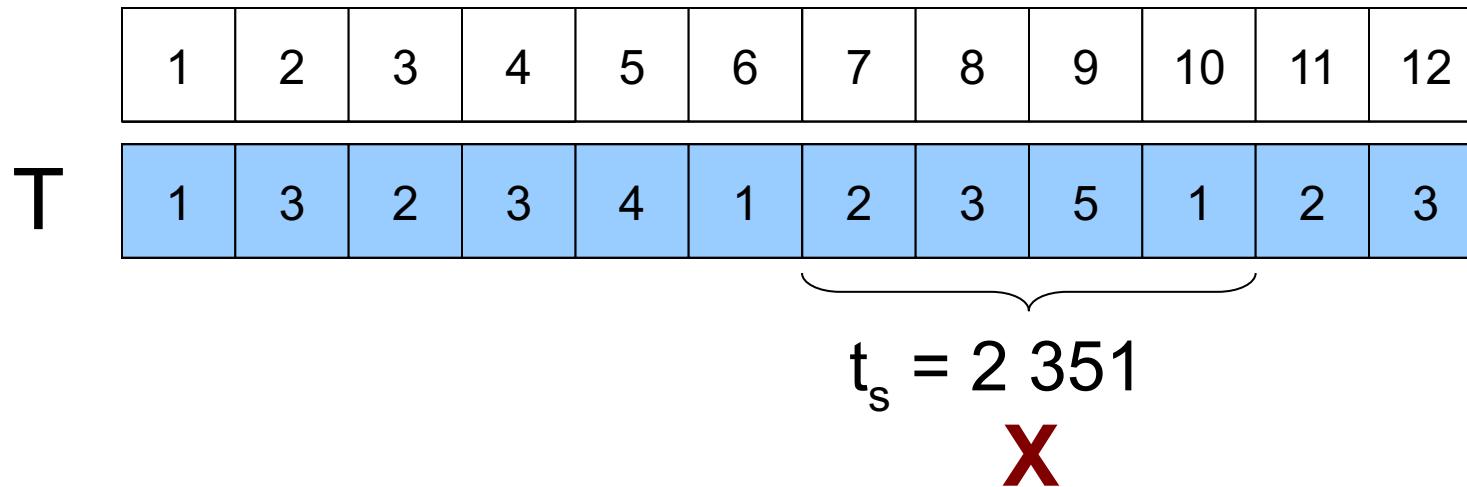
$p = 1\ 235$

# Rabin-Karp: formulation 1

---

$s=6$

$S=\{5\}$



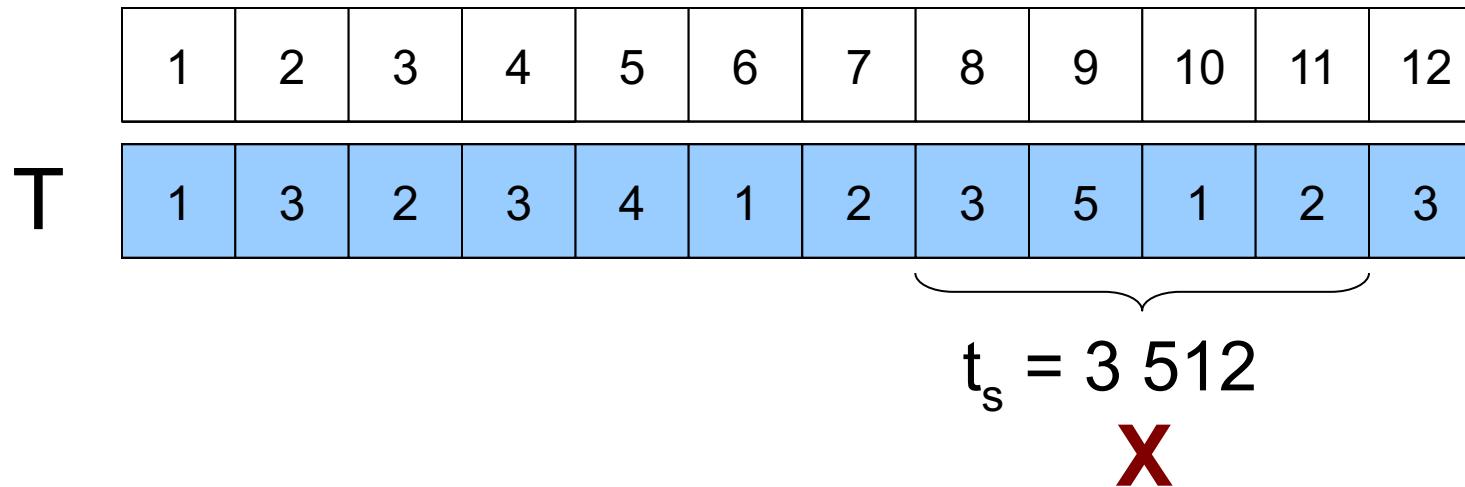
$p = 1\ 235$

# Rabin-Karp: formulation 1

---

$s=7$

$S=\{5\}$



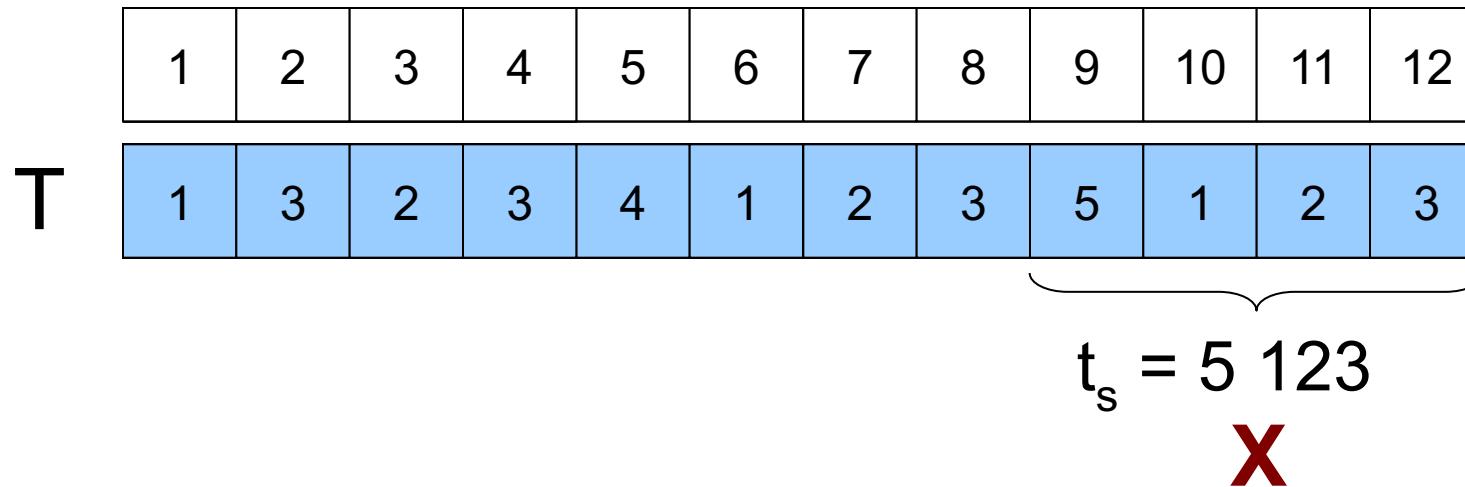
$p = 1235$

# Rabin-Karp: formulation 1

---

$s=8=n-m$

$S=\{5\}$



$p = 1 \ 235$

# Rabin-Karp: formulation 1

---

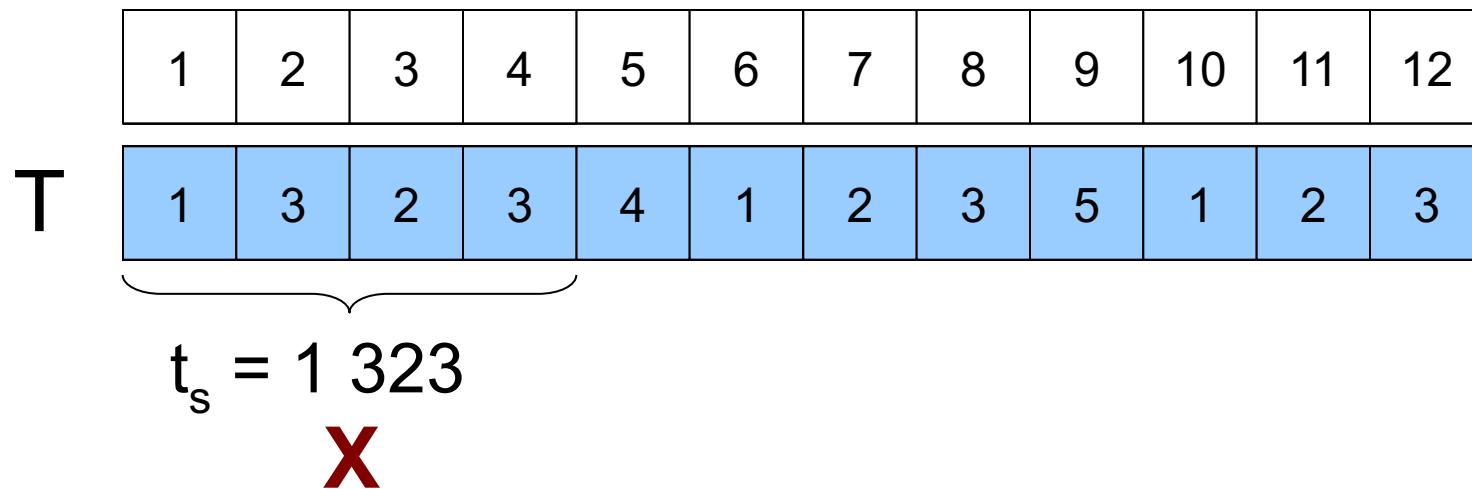
Réutiliser les résultats précédent:  
accélération

$$t_{s+1} = d(t_s - d^{m-1}T[s]) + T[m+s+1]$$

# Rabin-Karp: formulation 1

---

$s=0$



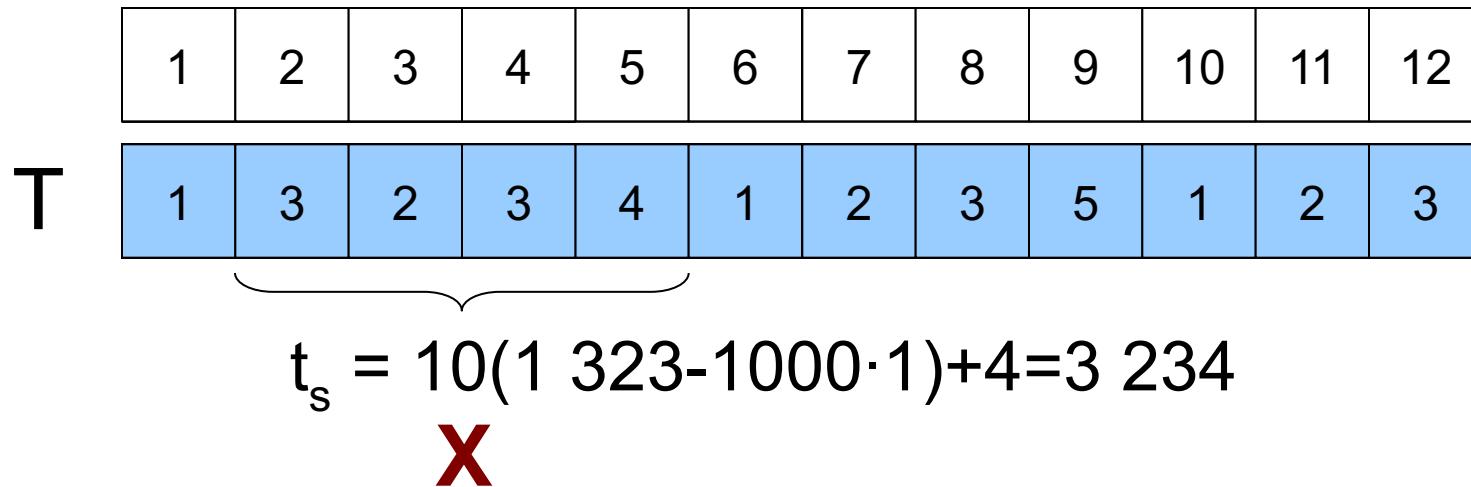
$p = 1\ 235$

---

# Rabin-Karp: formulation 1

---

s=1

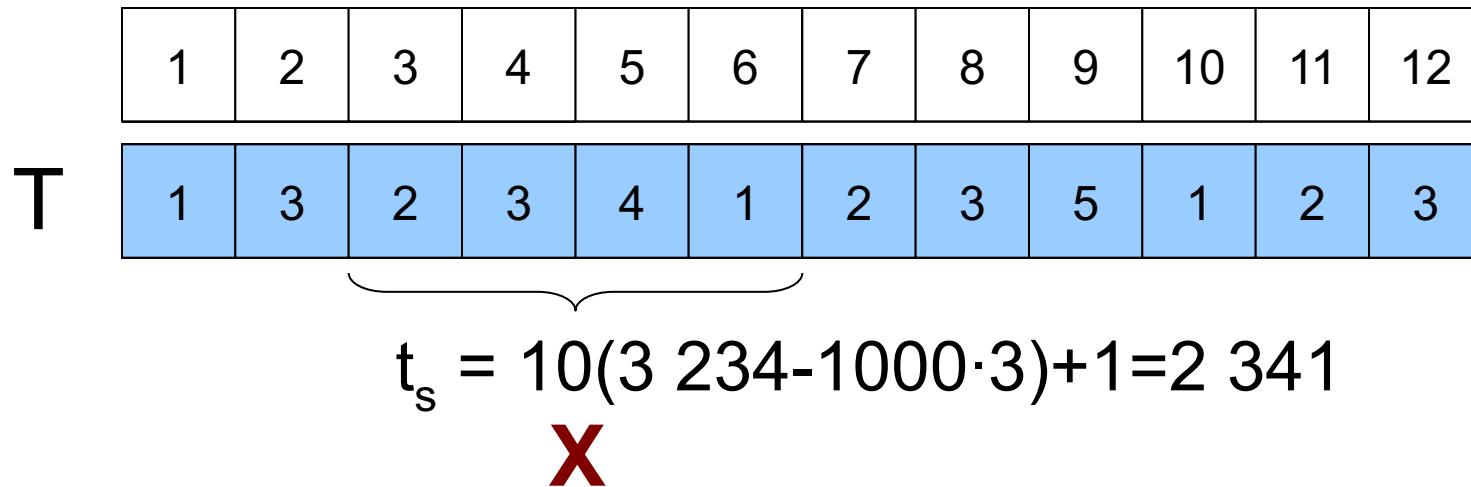


p = 1 235

# Rabin-Karp: formulation 1

---

s=2



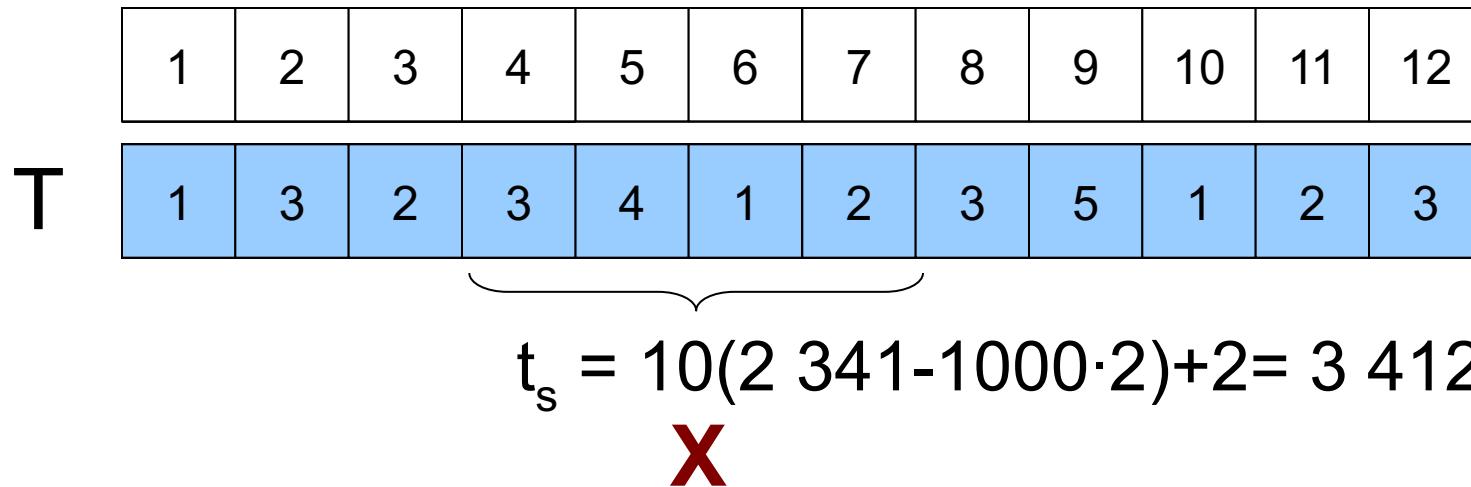
p = 1 235

---

# Rabin-Karp: formulation 1

---

$s=3$

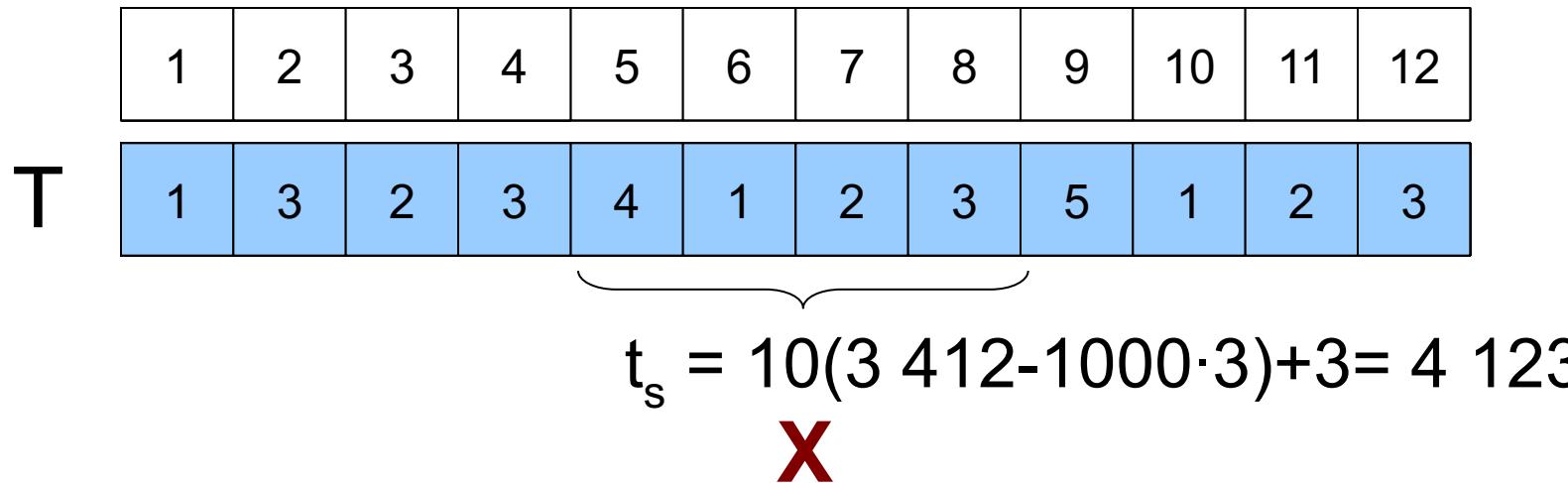


$p = 1 \ 235$

# Rabin-Karp: formulation 1

---

s=4



p = 1 235

---

# Rabin-Karp: formulation 1

---

s=5

S={5}

1	2	3	4	5	6	7	8	9	10	11	12	
T	1	3	2	3	4	1	2	3	5	1	2	3

$$t_s = 10(4 \cdot 123 - 1000 \cdot 4) + 5 = 1 \ 235$$



p = 1 235

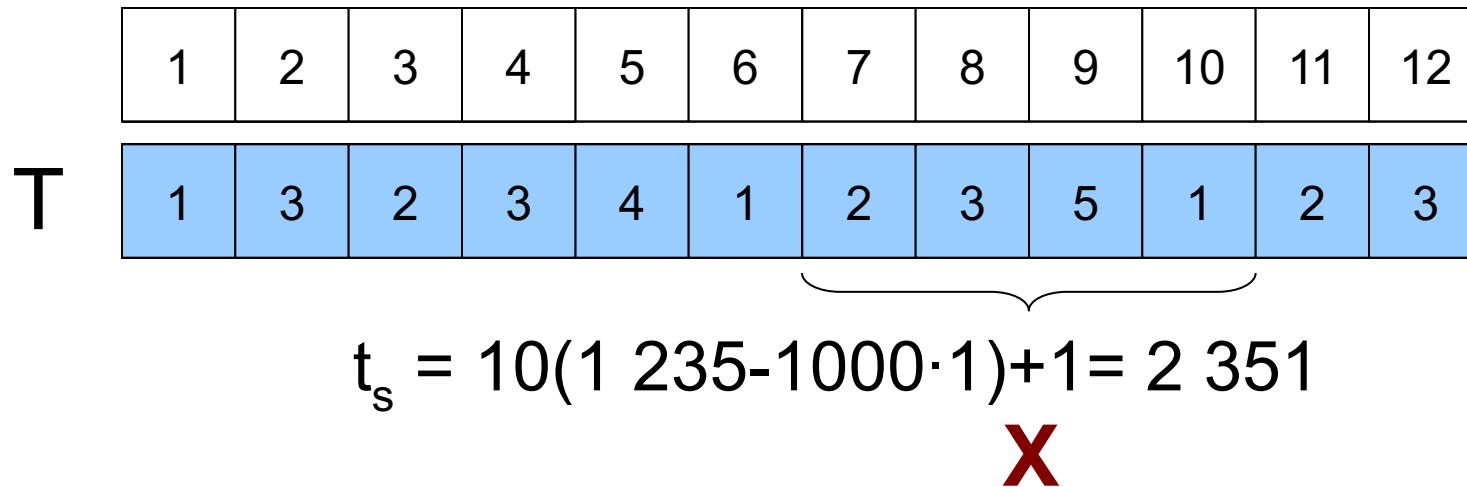
---

# Rabin-Karp: formulation 1

---

s=6

S={5}



p = 2 351

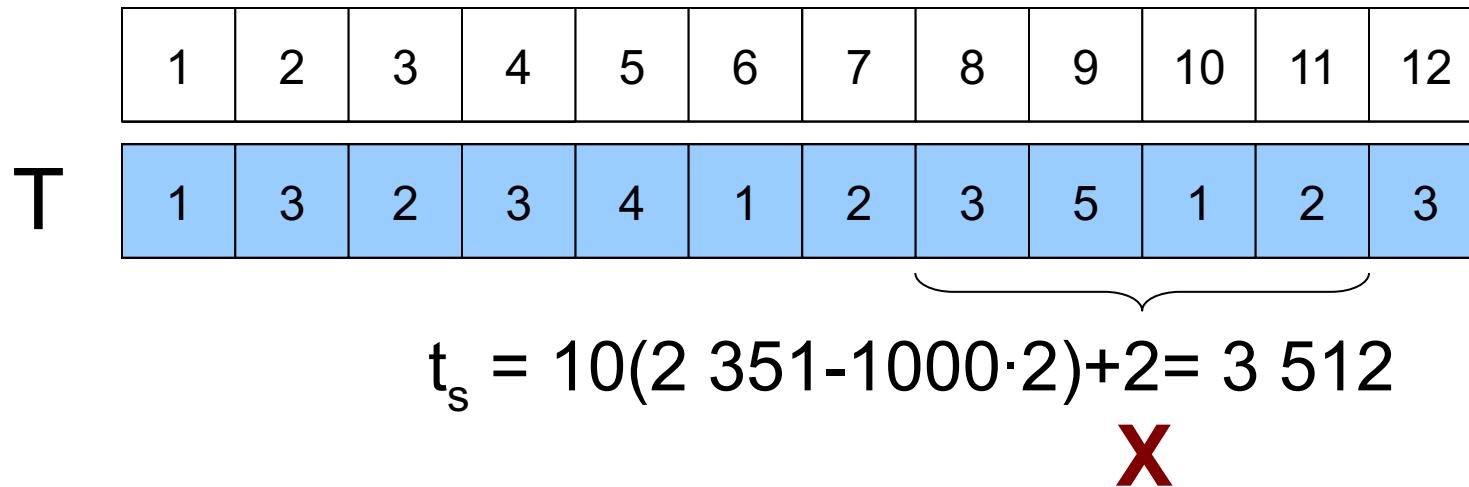
---

# Rabin-Karp: formulation 1

---

s=7

S={5}



p = 1 235

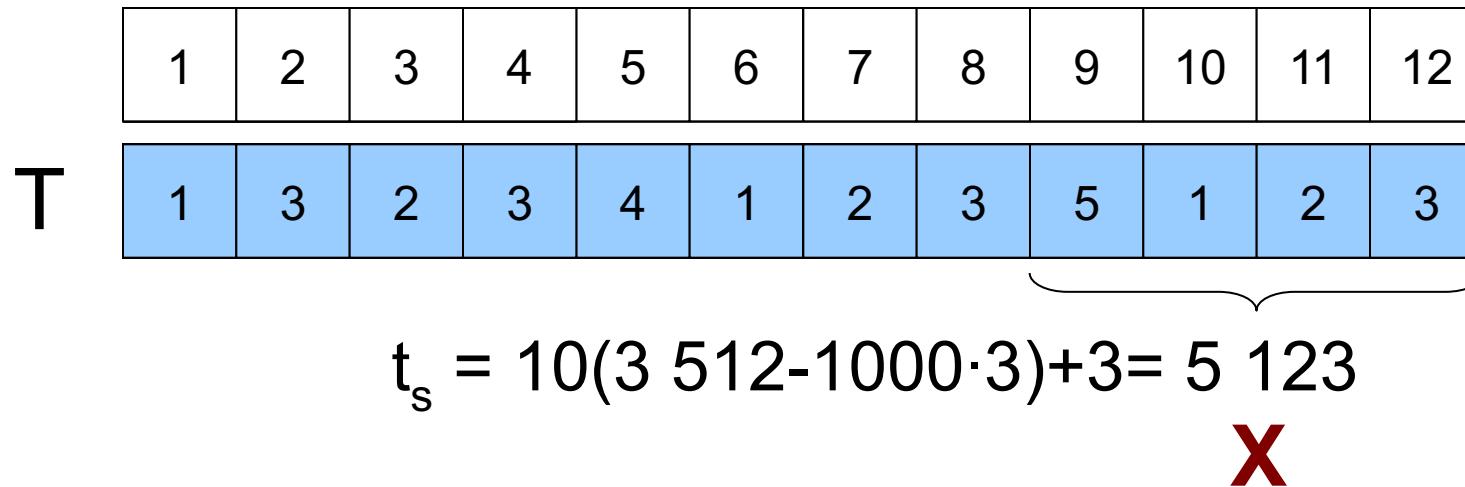
---

# Rabin-Karp: formulation 1

---

$s=8=n-m$

$S=\{5\}$



$p = 1 \ 235$

# Rabin-Karp: formulation 1

---

Problème:

L'alphabet  $\Sigma$  n'est pas celui des chiffres 0-9 et la base  $d=|\Sigma| \neq 10$

ASCII étendu : 256 caractères  $d = 256$

UNICODE: 65536 caractères  $d = 65536$

Utiliser Horner et la technique itérative sur  $t_s$  risque de poser problème quand même. Les nombres vont être grands...

# Rabin-Karp: formulation 2

---

Idée:

Utiliser une écriture modulaire. Pour un alphabet  $\Sigma$ , écrire  $P$  sous forme d'un nombre  $p$  dans la base  $d=|\Sigma|$  modulo  $q$

Choix de  $q$ :

On prend  $q$  de telle sorte que  $d \cdot q$  tiennent dans un mot machine (32 bits) :

**Raison:** essayer d'exprimer  $p$  en fonction de Horner...

# Rabin-Karp: formulation 2

---

Pour illustrer l'algorithme, reprenons l'exemple, on utilise les chiffre 0-9: on traduit P sous la forme du nombre p dans la base  $d=|\Sigma|=10$  modulo q = 11

P    

1	2	3	5
---	---	---	---

Exemple:

$$p = \{[\{[1 \cdot 10 + 2] \text{mod } 11] \cdot 10 + 3\} \text{mod } 11] \cdot 10 + 5\} \text{mod } 11$$

$$p = \{[\{[12] \text{mod } 11] \cdot 10 + 3\} \text{mod } 11] \cdot 10 + 5\} \text{mod } 11$$

$$p = \{[\{[1] \cdot 10 + 3\} \text{mod } 11] \cdot 10 + 5\} \text{mod } 11$$

$$p = \{[2] \cdot 10 + 5\} \text{mod } 11$$

$$p = \{25\} \text{mod } 11 = 3$$

# Rabin-Karp: formulation 2

---

Réutiliser les résultats précédent:  
accélération

$$t_{s+1} = \{d(t_s - hT[s]) + T[m+s+1]\} \bmod q$$

$$h = d^{m-1} \bmod q$$

# Rabin-Karp: formulation 2

---

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

T	1	3	2	3	4	1	2	3	5	1	2	3
---	---	---	---	---	---	---	---	---	---	---	---	---

P	1	2	3	5
---	---	---	---	---

$$p = 1235 \bmod 11 = 3$$

# Rabin-Karp: formulation 2

---

s=0

1	2	3	4	5	6	7	8	9	10	11	12	
T	1	3	2	3	4	1	2	3	5	1	2	3

$$t_s = 1 \ 323 \text{mod} \ 11 = 3$$

?

p = 3

---

# Rabin-Karp: formulation 2

---

s=0

1	2	3	4	5	6	7	8	9	10	11	12	
T	1	3	2	3	4	1	2	3	5	1	2	3

$t_s = 1 \ 323 \text{mod} \ 11 = 3$

**faux-positif**

p = 3

# Rabin-Karp: formulation 2

---

s=0

1	2	3	4	5	6	7	8	9	10	11	12	
T	1	3	2	3	4	1	2	3	5	1	2	3

$$t_s = 1323 \bmod 11 = 3$$

X

p = 3

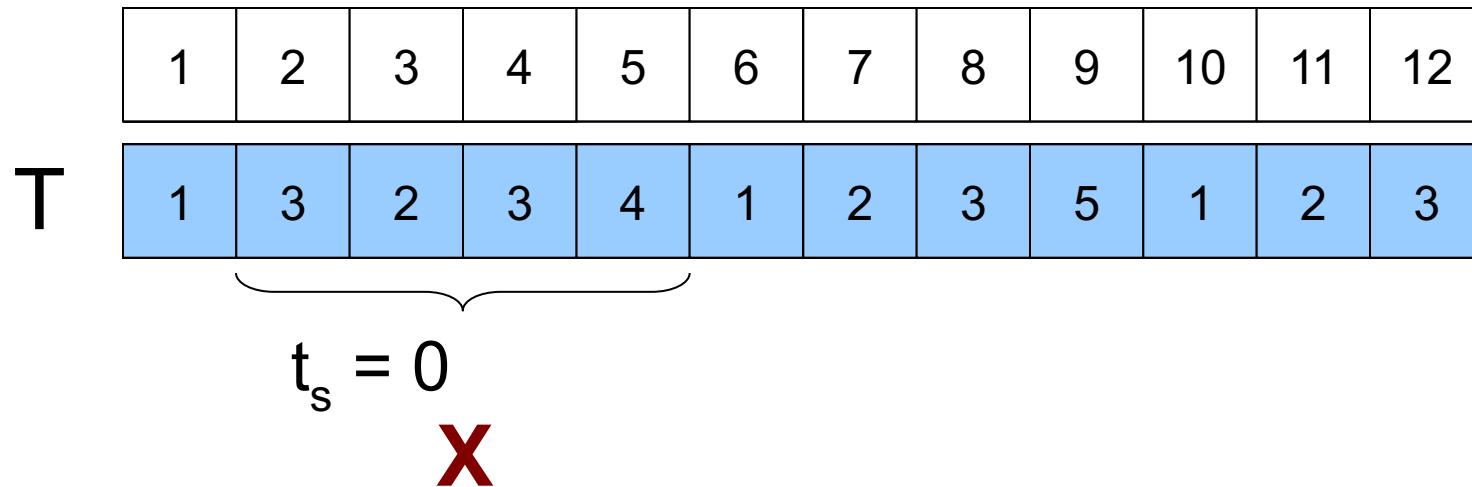
P

1	2	3	5
---	---	---	---

# Rabin-Karp: formulation 2

---

$s=1$



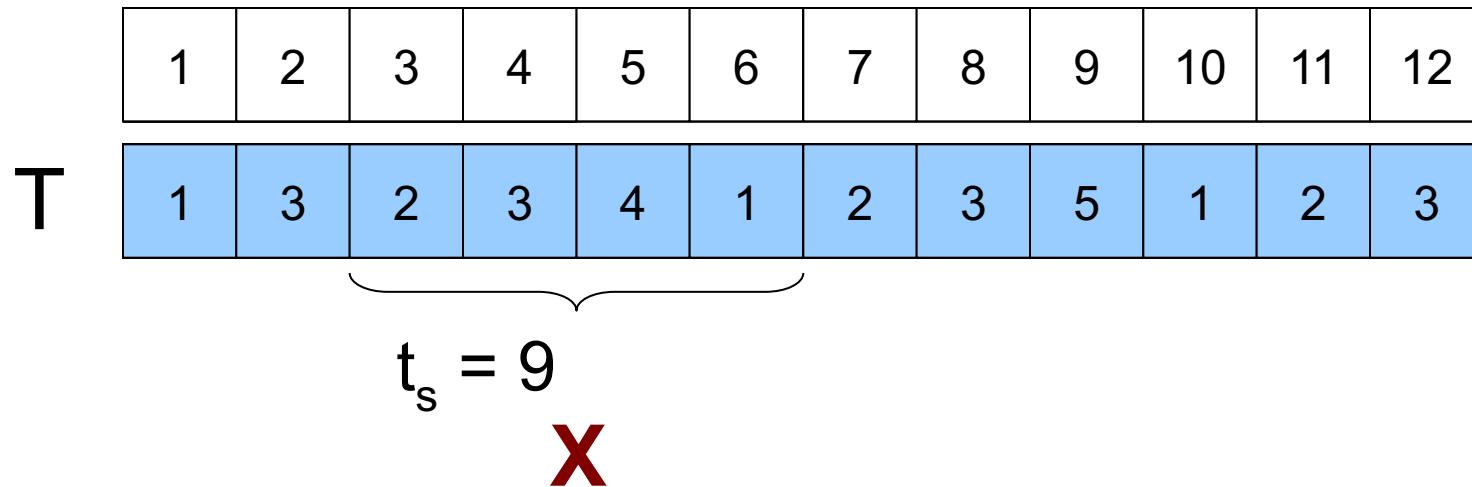
$p = 3$

---

# Rabin-Karp: formulation 2

---

$s=2$



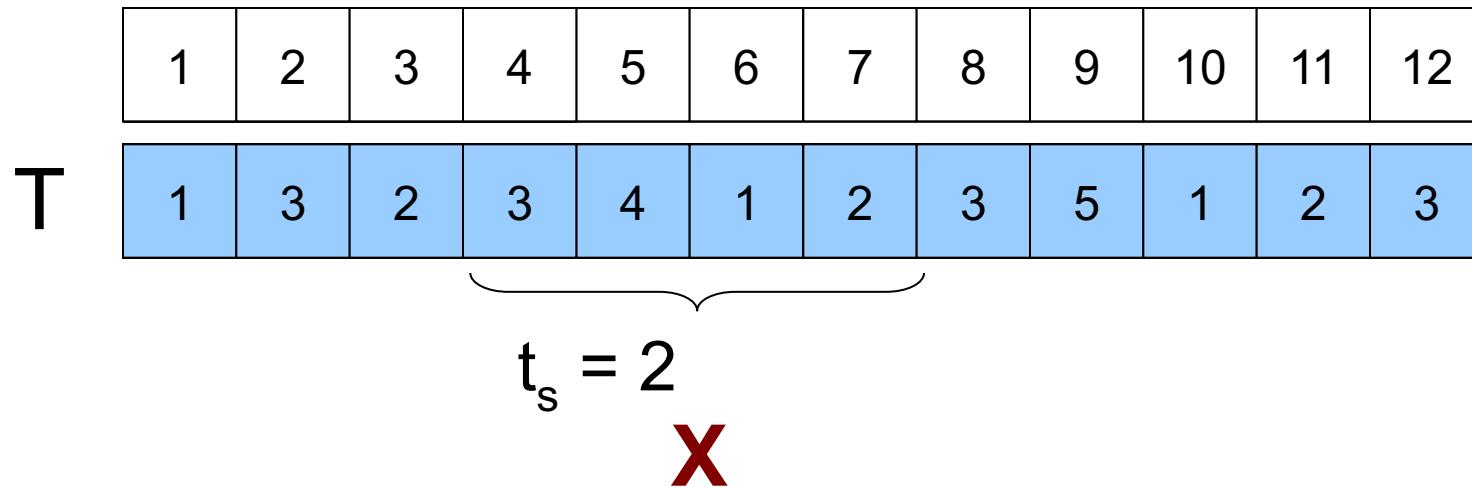
$p = 3$

---

# Rabin-Karp: formulation 2

---

$s=3$

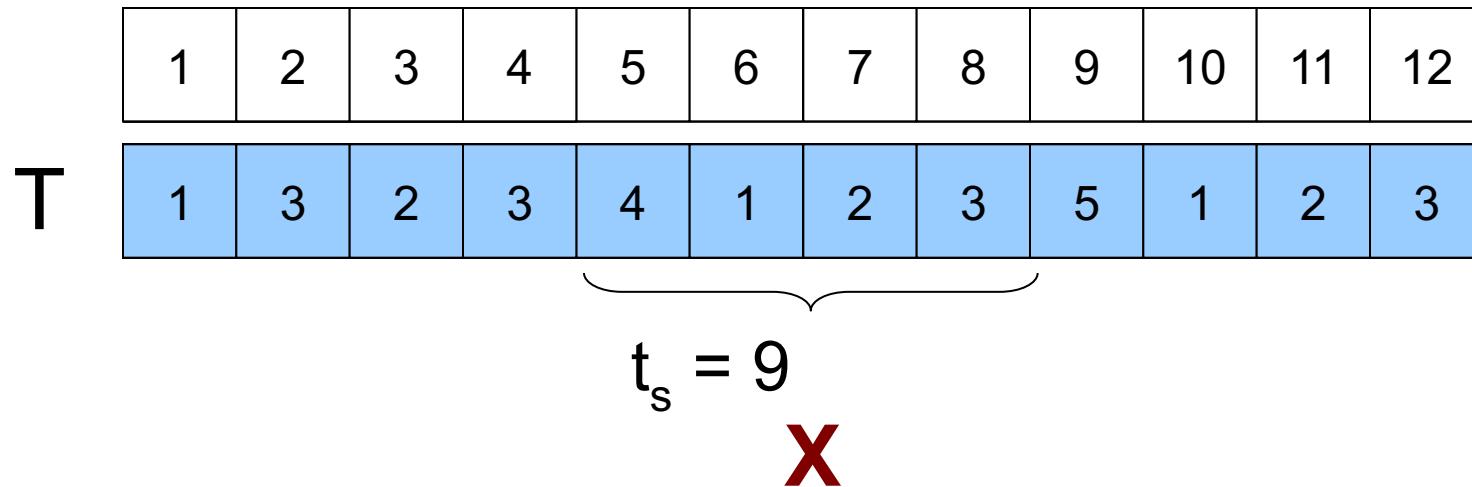


$p = 3$

# Rabin-Karp: formulation 2

---

$s=4$



$p = 3$

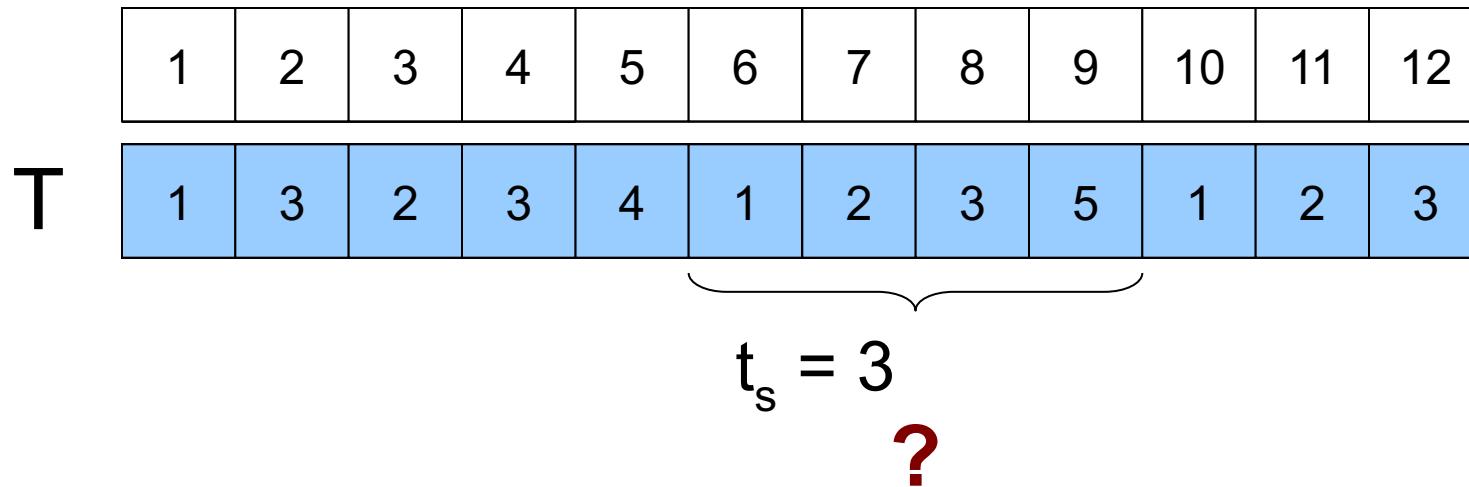
---

# Rabin-Karp: formulation 2

---

$s=5$

$S=\{5\}$



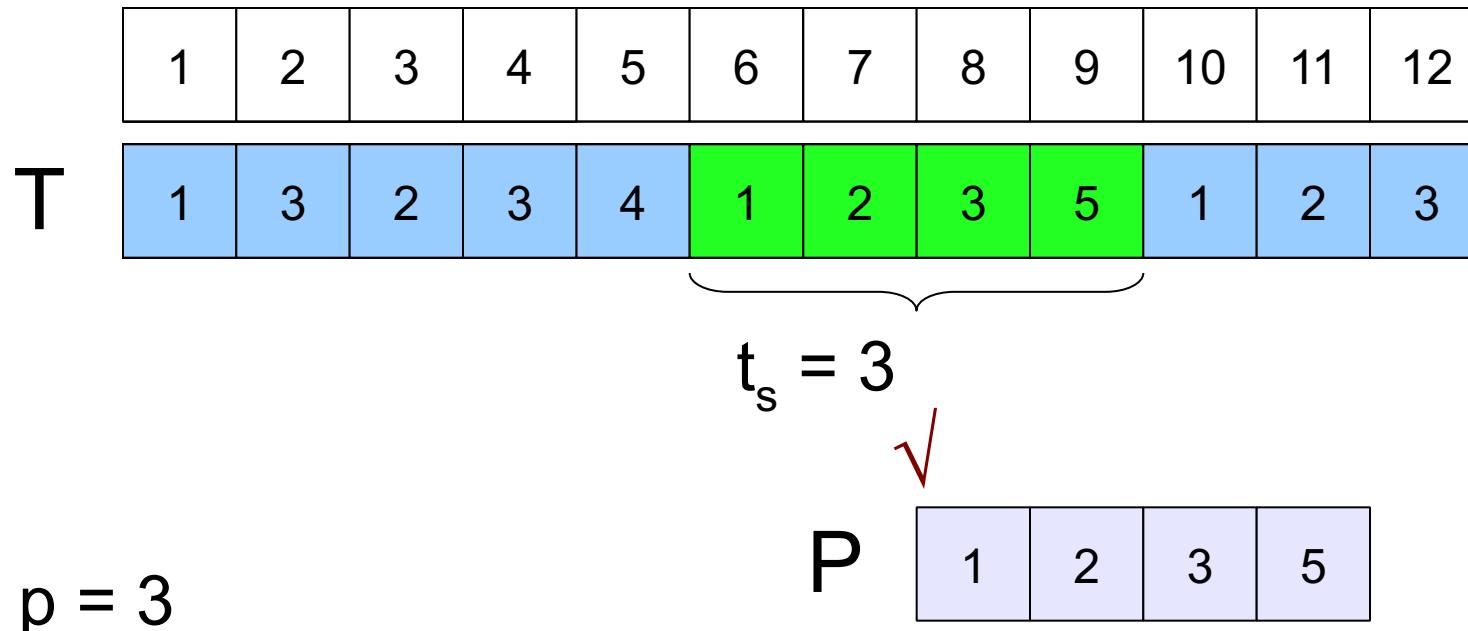
$p = 3$

# Rabin-Karp: formulation 2

---

$s=5$

$S=\{5\}$

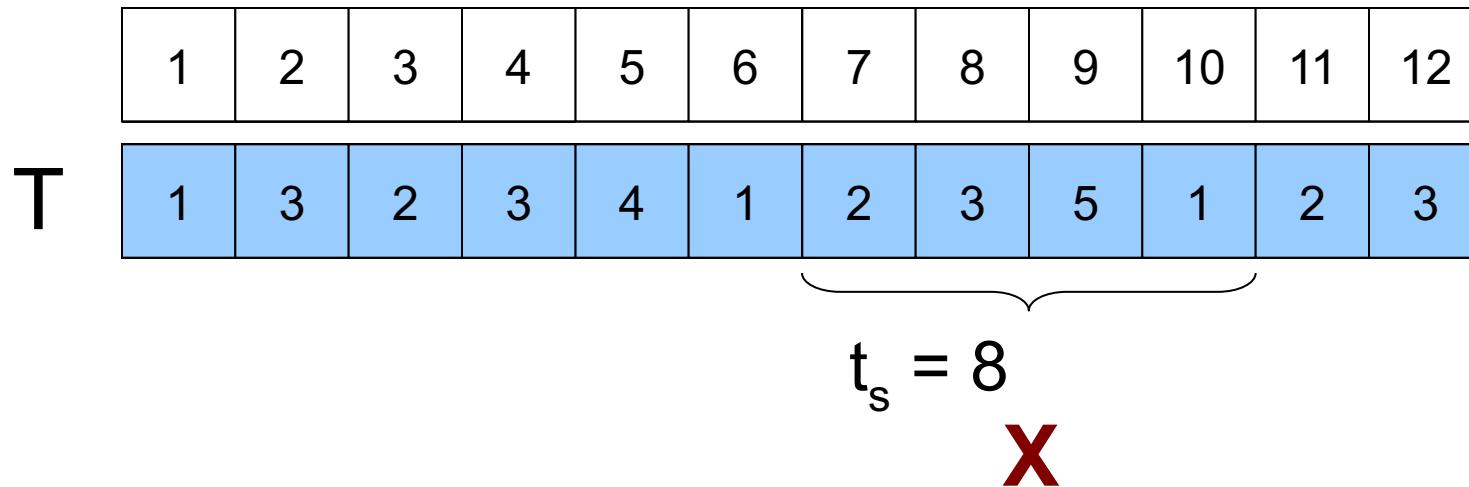


# Rabin-Karp: formulation 2

---

$s=6$

$S=\{5\}$



$p = 3$

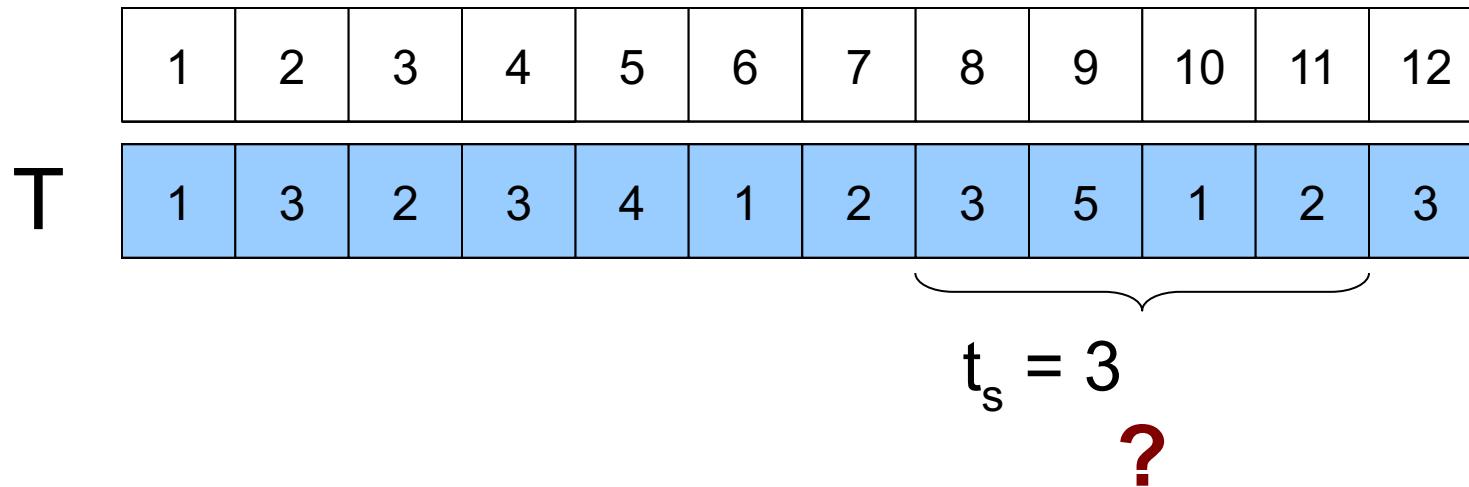
---

# Rabin-Karp: formulation 2

---

$s=7$

$S=\{5\}$



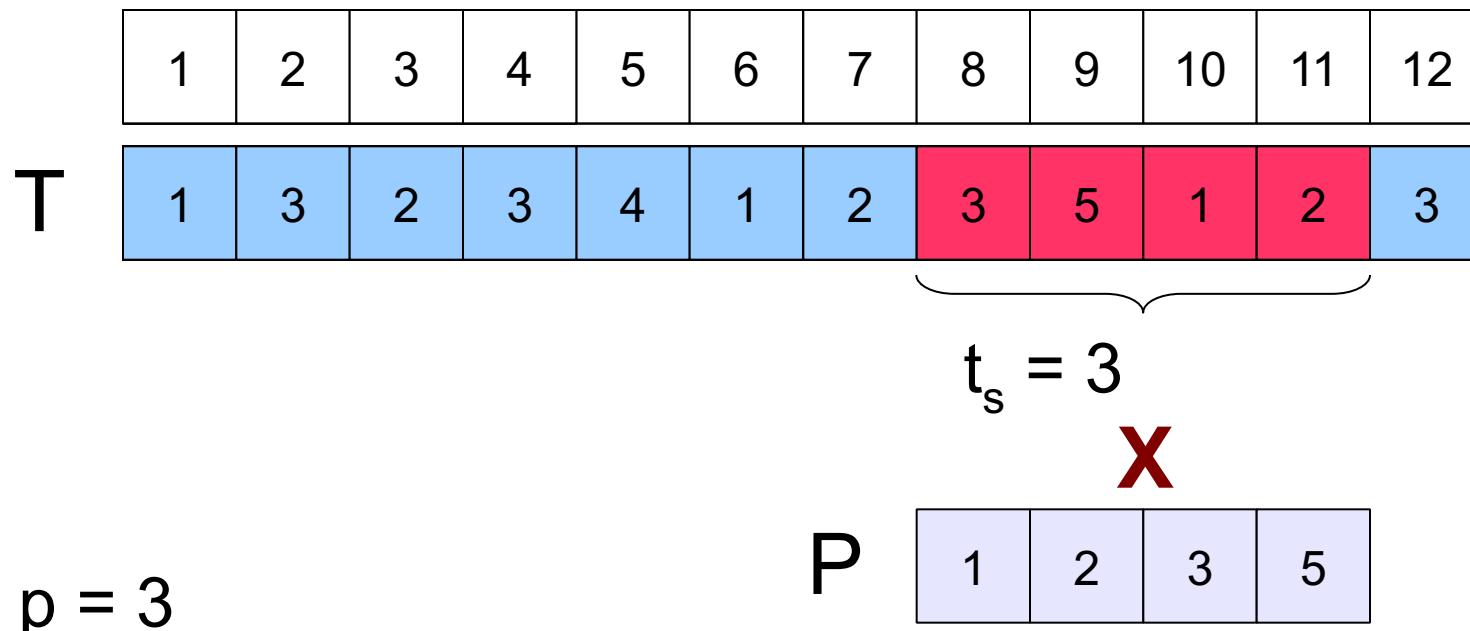
$p = 3$

# Rabin-Karp: formulation 2

---

$s=7$

$S=\{5\}$



---

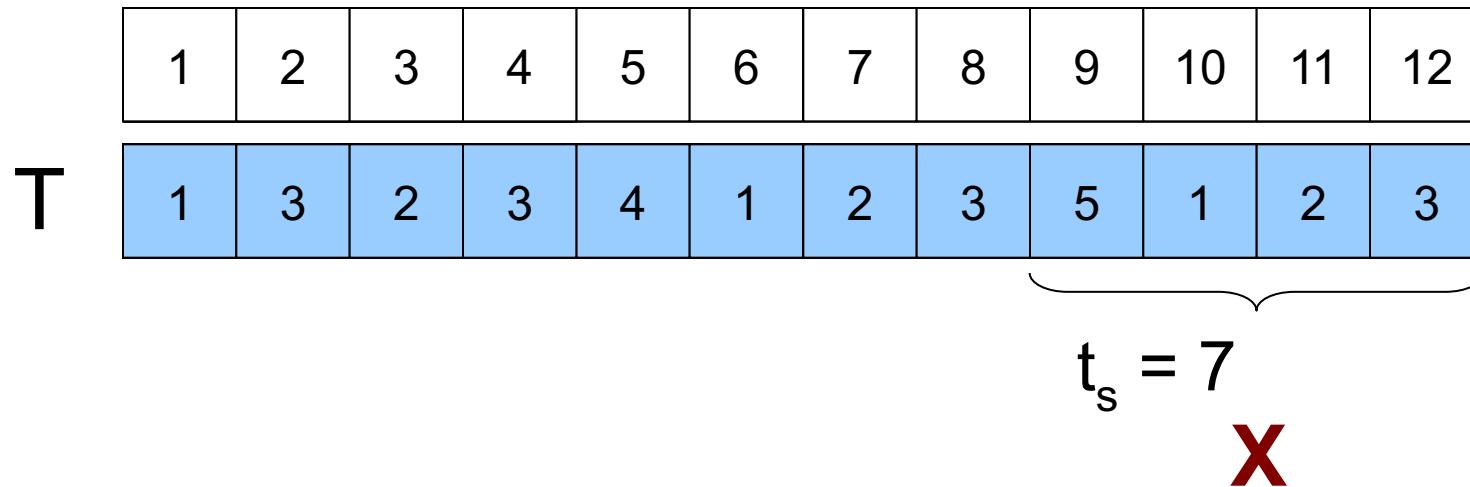
$p = 3$

# Rabin-Karp: formulation 2

---

$s=8=n-m$

$S=\{5\}$

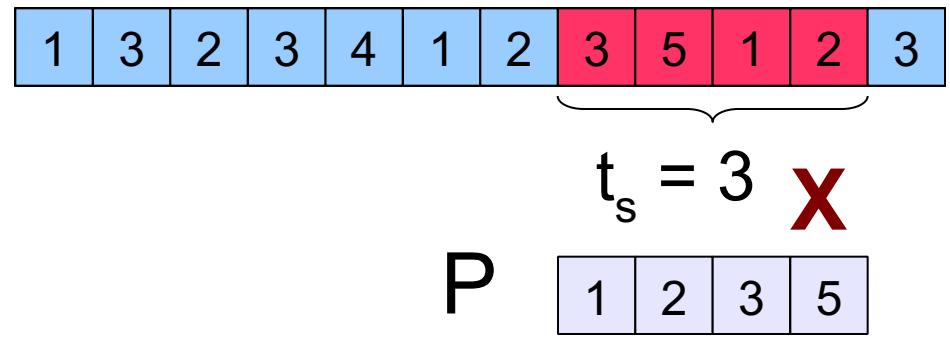


$p = 3$

# Rabin-Karp

Complexité de l'algorithme à cause des faux-positifs

$O(m(n-m+1))$



$O(n)$  en cas moyen et en meilleur cas

Néanmoins, l'exécution de Rabin Karp est **plus rapide** que l'algorithme naïf.

# Plan

---

Recherche de patron

Problématique

Rabin Karp

Automate FSM

PLSC

Problématique

Solution par programmation dynamique

# Automate FSM

---

Objectif: meilleure complexité

Battre  $O(m(n-m+1))$

Meilleure performances que Rabin-Karp

Analyser la chaîne de caractère:

Construire une machine à états (prétraitement)

Réutiliser les résultats précédents:

L'automate possède une mémoire interne de son état (accélération)

# Automate FSM

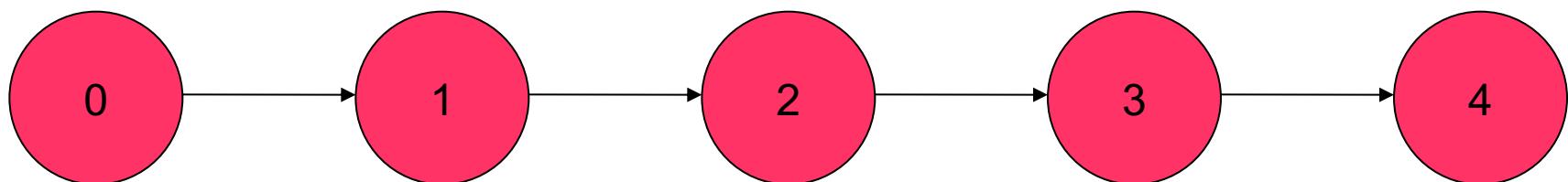
---

**Construire - Le nombre d'états est égal à m+1**

Exemple:

P    

b	e	b	e
---	---	---	---



# Automate FSM

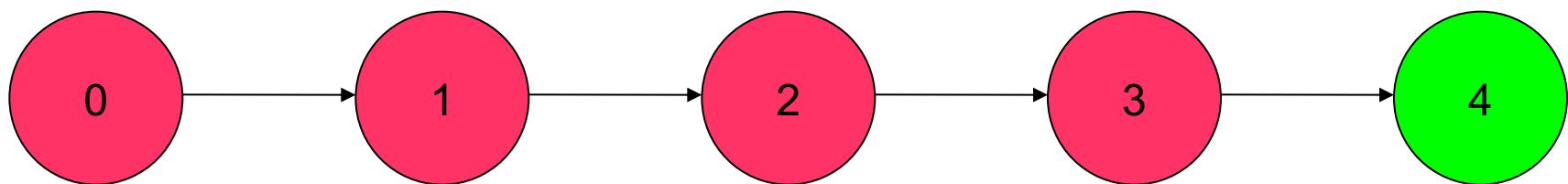
---

**Construire - Le dernier état valide l'entrée**

Exemple:

P    

b	e	b	e
---	---	---	---



# Automate FSM

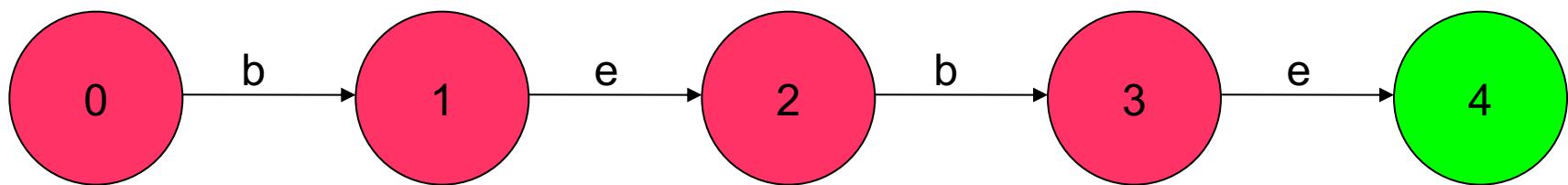
---

**Construire – Les arcs répondent aux unités**

Exemple:

P    

b	e	b	e
---	---	---	---



# Automate FSM

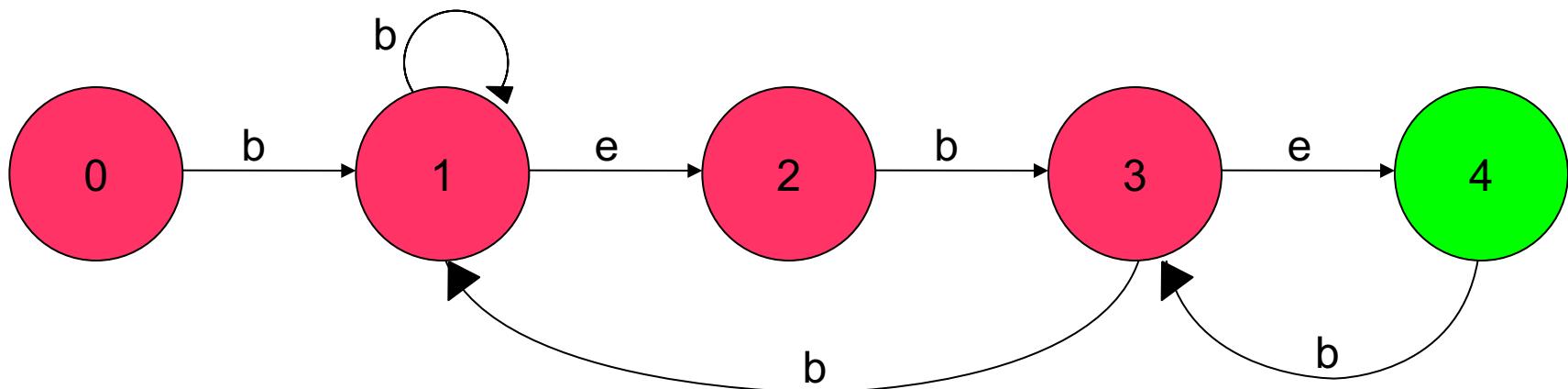
**ConstruRe** – Ajouter les arcs restants

Exemple:

P    

b	e	b	e
---	---	---	---

**Note:** les arcs absents mènent à l'état de départ (0)

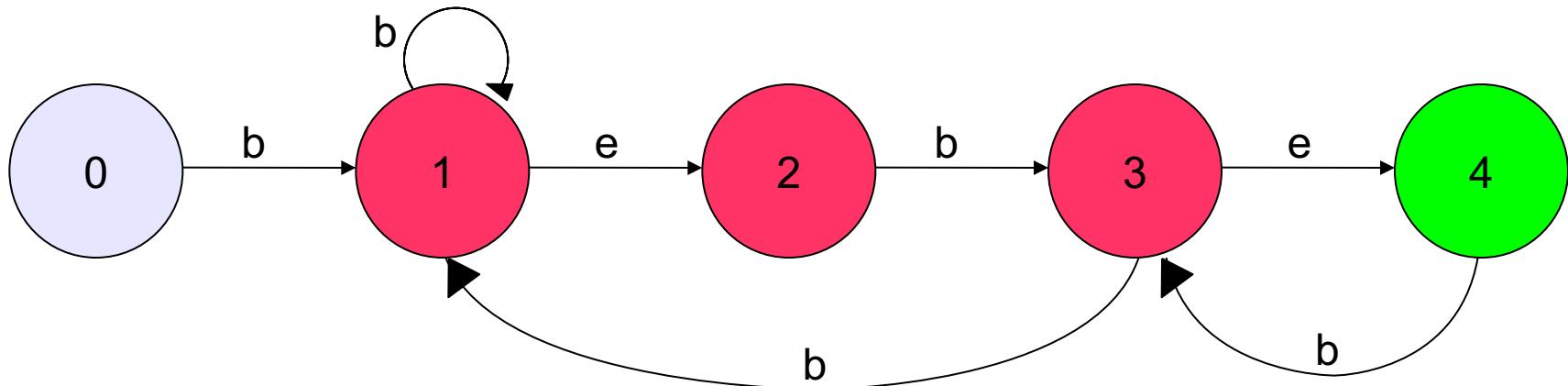


# Automate FSM

Traîter – La chaîne T est traversée une fois

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

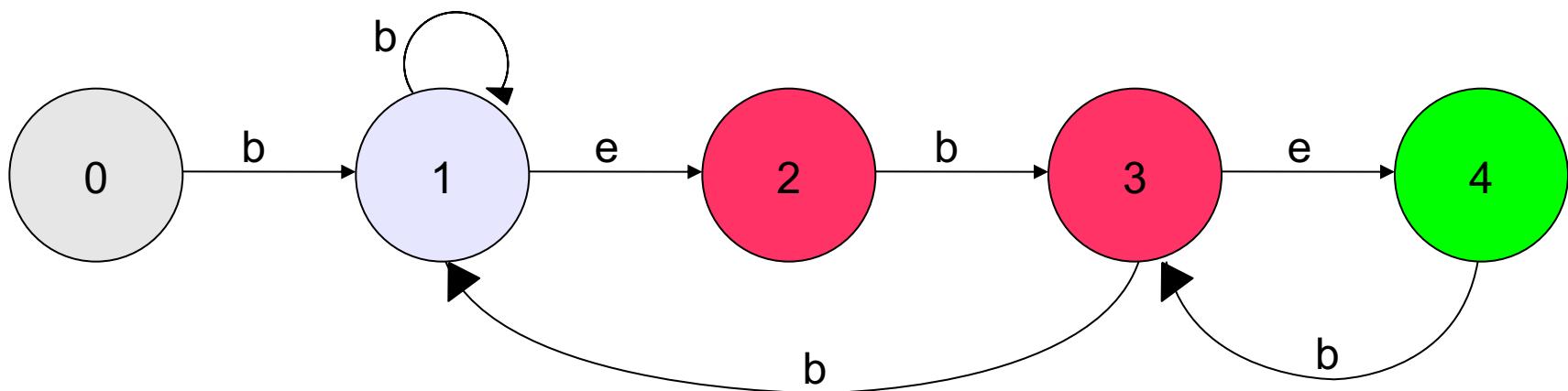
T	b	e	e	b	b	e	b	e	b	e	e	b
---	---	---	---	---	---	---	---	---	---	---	---	---



# Automate FSM

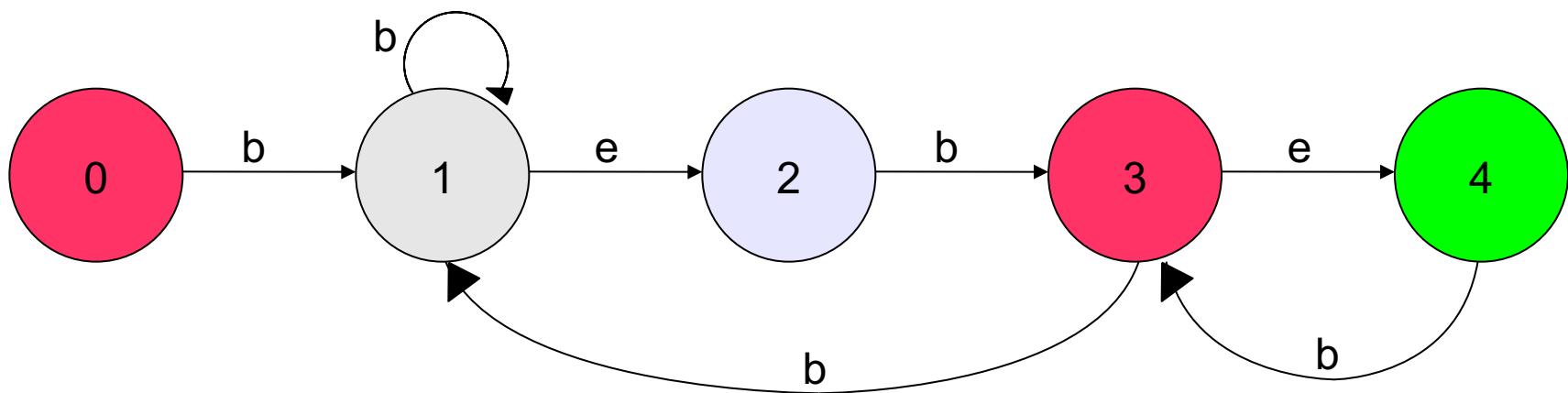
---

1	2	3	4	5	6	7	8	9	10	11	12	
T	b	e	e	b	b	e	b	e	b	e	e	b



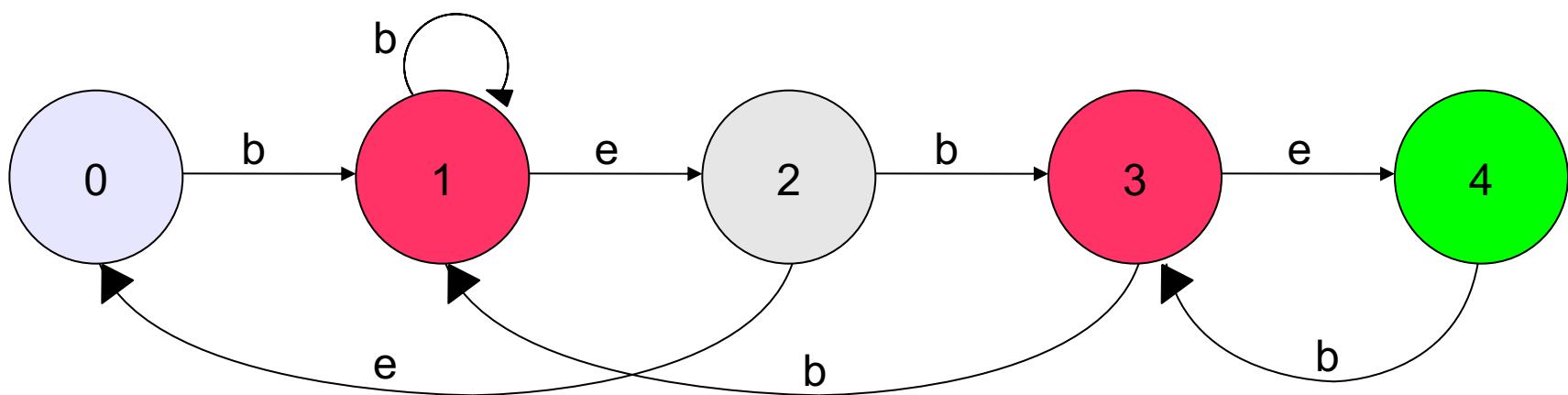
# Automate FSM

1	2	3	4	5	6	7	8	9	10	11	12	
T	b	e	e	b	b	e	b	e	b	e	e	b



# Automate FSM

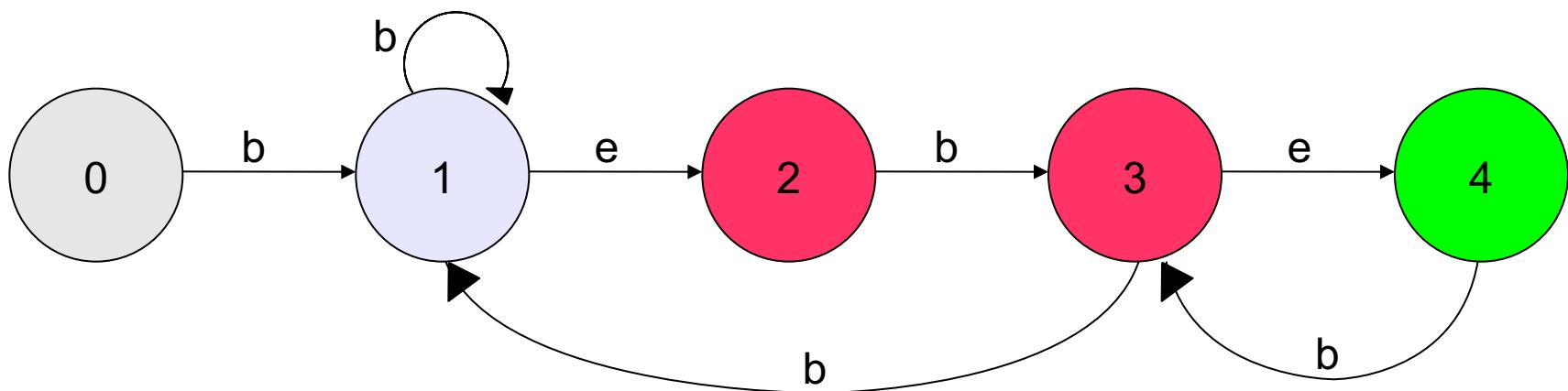
1	2	3	4	5	6	7	8	9	10	11	12	
T	b	e	e	b	b	e	b	e	b	e	e	b



# Automate FSM

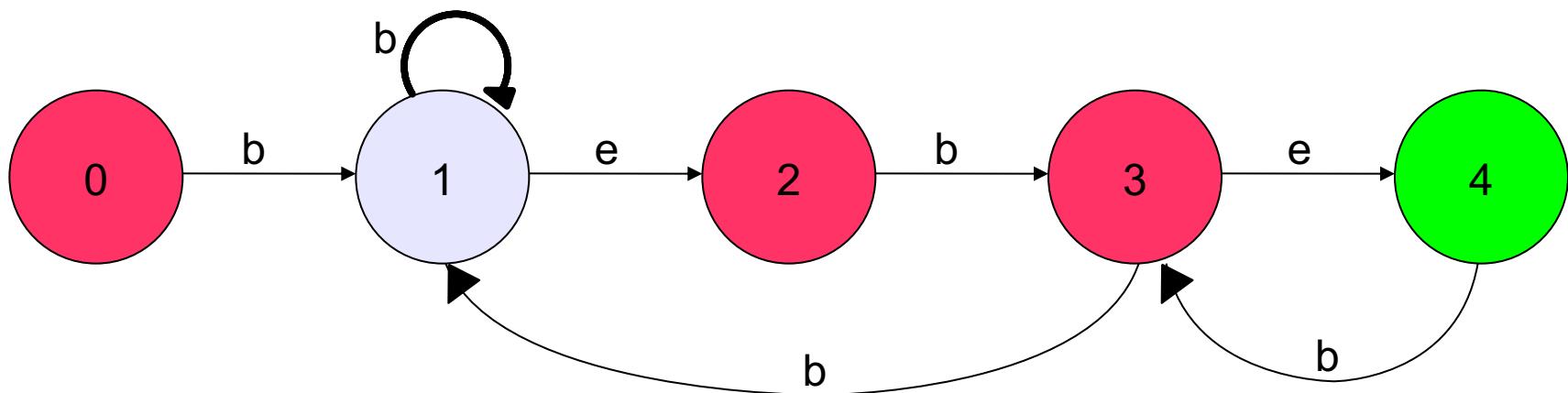
---

1	2	3	4	5	6	7	8	9	10	11	12	
T	b	e	e	b	b	e	b	e	b	e	e	b



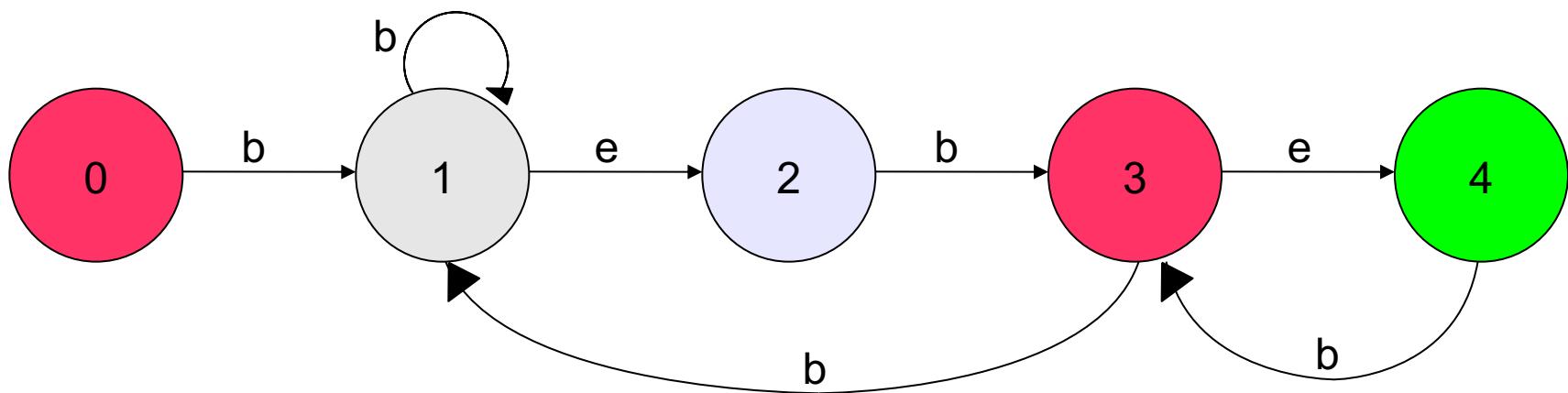
# Automate FSM

1	2	3	4	5	6	7	8	9	10	11	12	
T	b	e	e	b	b	e	b	e	b	e	e	b



# Automate FSM

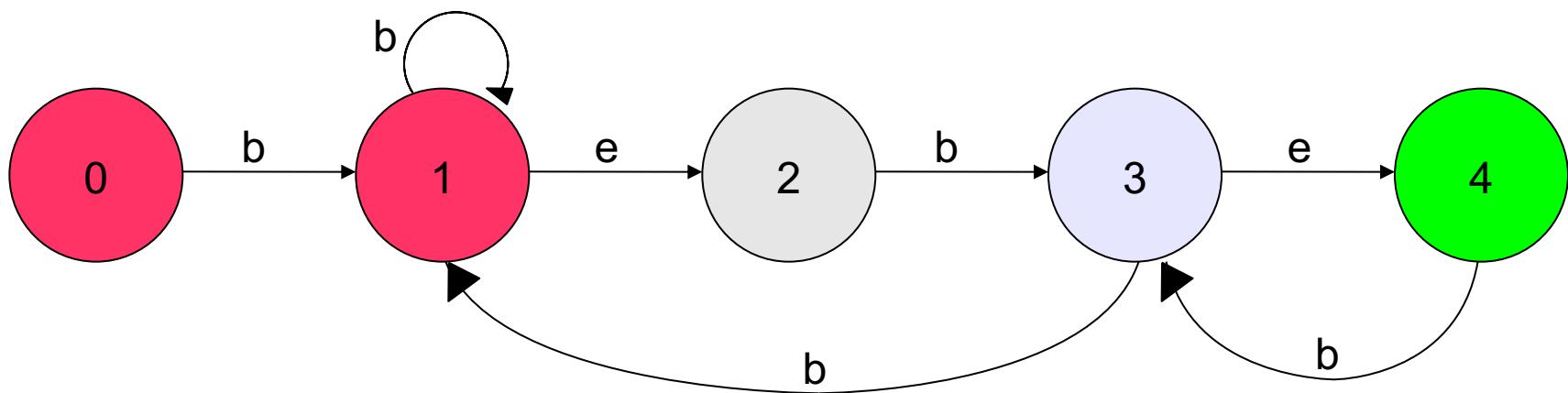
1	2	3	4	5	6	7	8	9	10	11	12	
T	b	e	e	b	b	e	b	e	b	e	e	b



# Automate FSM

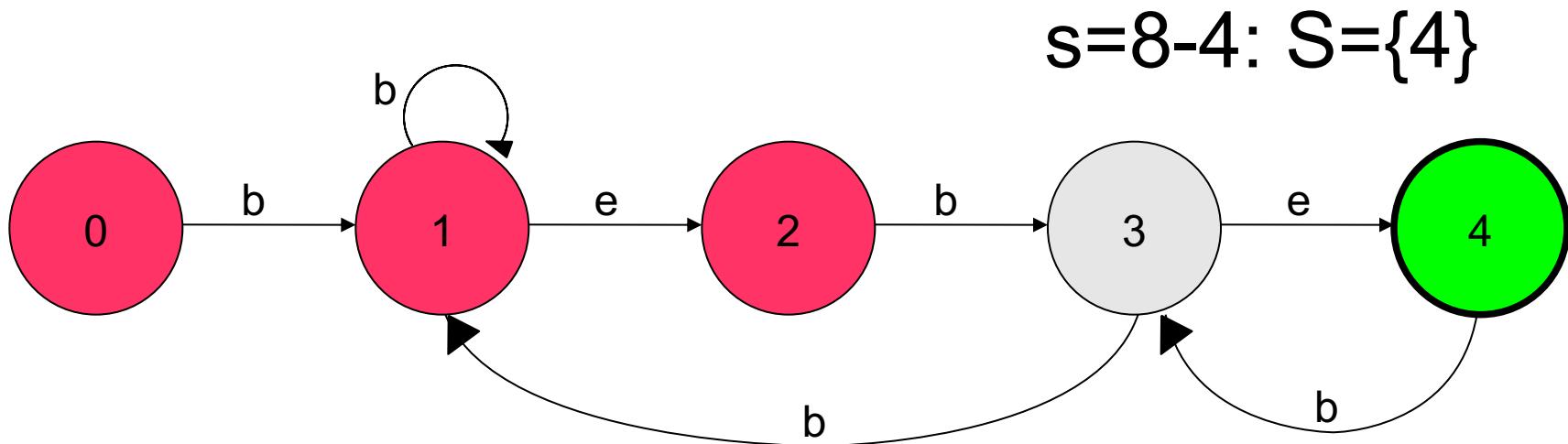
---

1	2	3	4	5	6	7	8	9	10	11	12	
T	b	e	e	b	b	e	b	e	b	e	e	b



# Automate FSM

1	2	3	4	5	6	7	8	9	10	11	12	
T	b	e	e	b	b	e	b	e	b	e	e	b

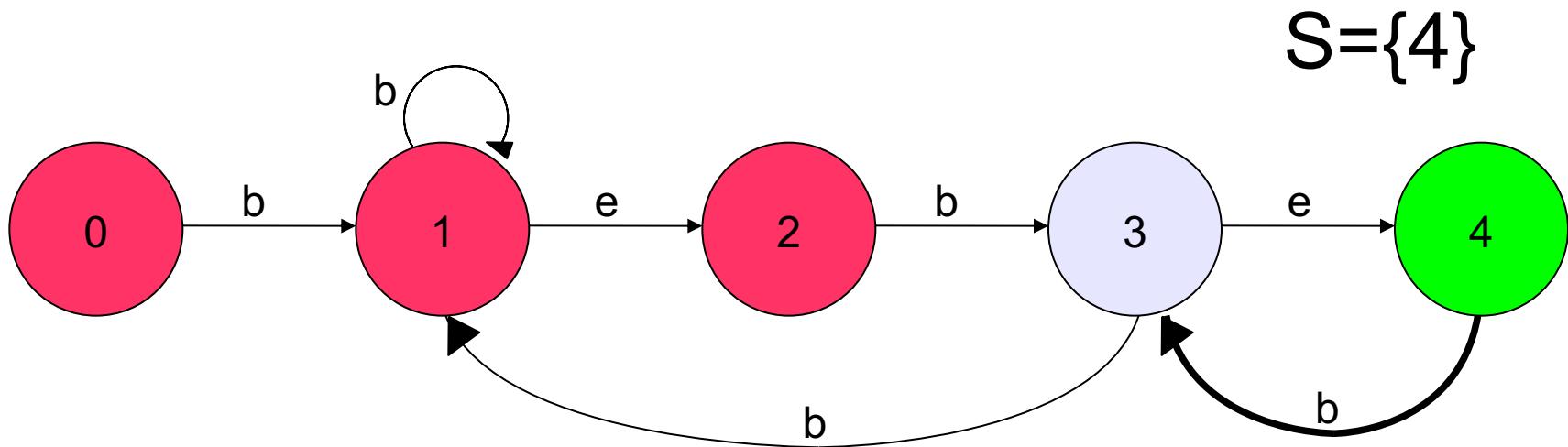


# Automate FSM

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

T

b	e	e	b	b	e	b	e	b	e	e	b
---	---	---	---	---	---	---	---	---	---	---	---

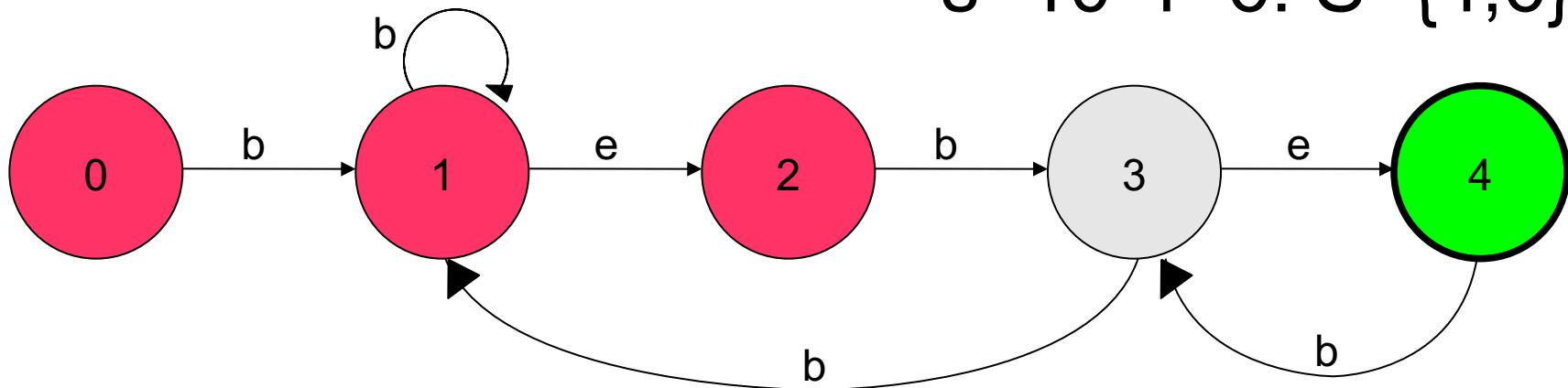


# Automate FSM

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

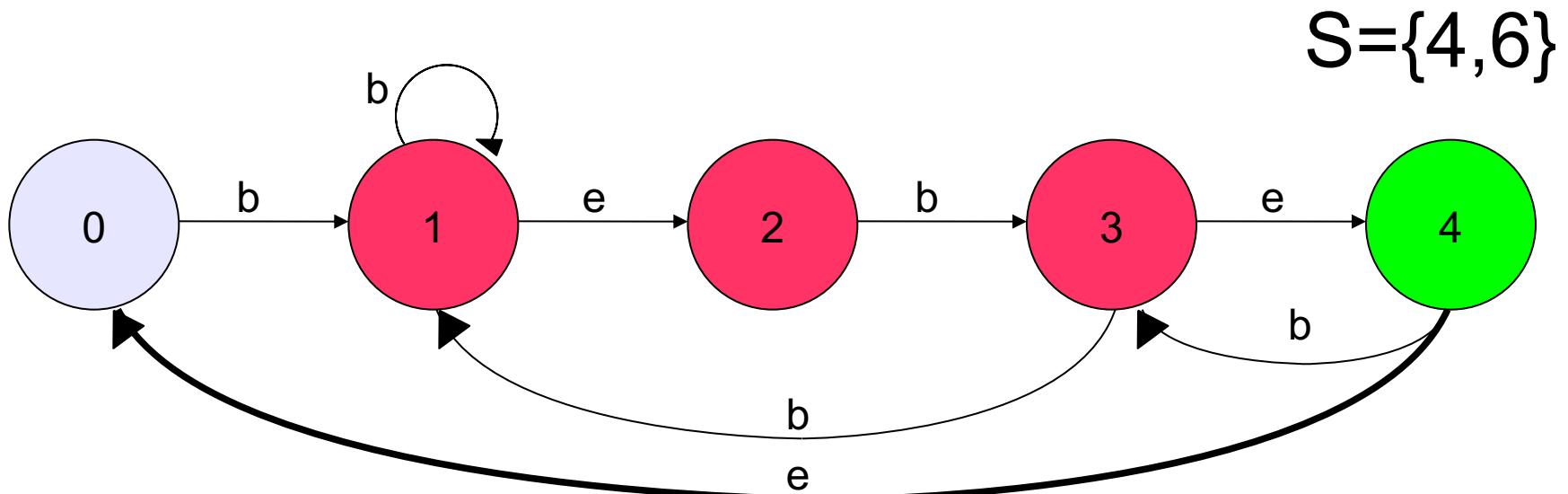
T	b	e	e	b	b	e	b	e	b	e	e	b
---	---	---	---	---	---	---	---	---	---	---	---	---

$$s=10-4=6: S=\{4,6\}$$



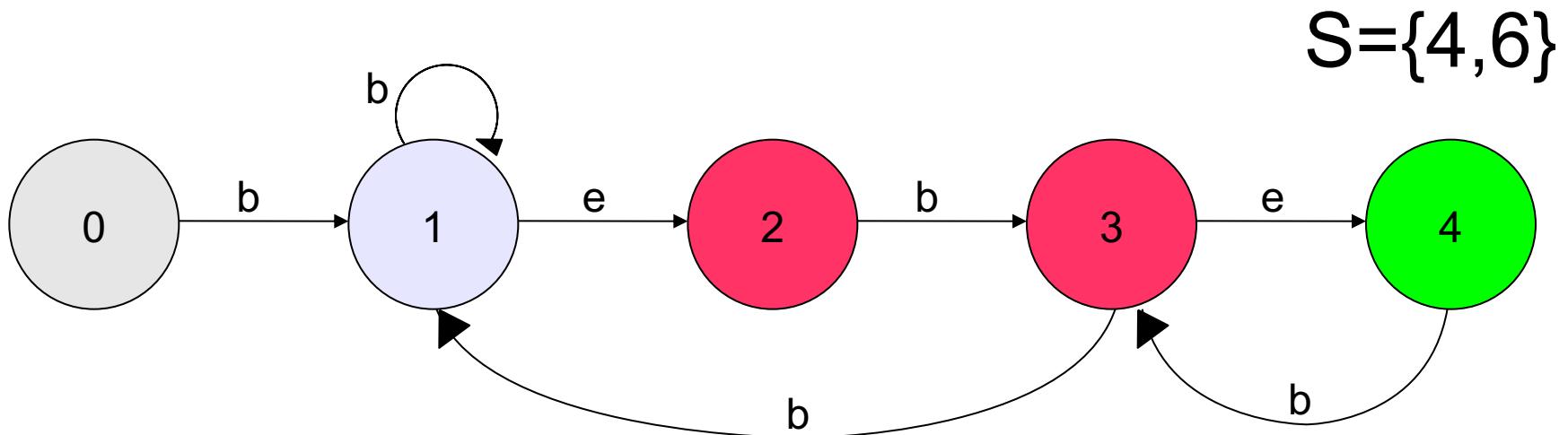
# Automate FSM

1	2	3	4	5	6	7	8	9	10	11	12
T	b	e	e	b	b	e	b	e	b	e	e



# Automate FSM

1	2	3	4	5	6	7	8	9	10	11	12	
T	b	e	e	b	b	e	b	e	b	e	e	b



# Automate FSM

---

Quelques définitions

Ensemble des états  $Q=\{q_0, q_1, q_2, \dots\}$

Les états peuvent être représentés par un entier (leur indice)

État initial: 0

État final: m

Fonction post-fixe est notée:  $\supset$

Fonction préfixe est notée:  $\subset$

Le préfixe de P de longueur i est noté  $P_i$  ( $P_i \subset P$ )

Alphabet:  $\Sigma$

Alphabet réduit aux lettres du patron:  $\Sigma_p$

Fonction de transition  $\delta : Q \times \Sigma_p \rightarrow Q$

# Construction

---

Algorithme de construction de la fonction de transition

Initialiser  $\delta$  à 0

**Pour**  $q = 0 : m$

**Pour** chaque caractère  $a$  dans  $\Sigma_P$

$k = \min(m+1, q+2)$

**Répéter**

$k = k - 1$

**Tant que**  $k > 0$  et (  $(P_k \supset P_q a)$  est faux)

$\delta(q, a) = k$

**Fin Pour**

**Fin Pour**

# Exemple

---

Rechercher: aabab

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

$$Q_0 = 0$$

État final : 5

$$\Sigma_p = \{a, b\}$$

$$q_0 \leftrightarrow \epsilon$$

$$q_1 \leftrightarrow a$$

$$q_2 \leftrightarrow aa$$

$$q_3 \leftrightarrow aab$$

$$q_4 \leftrightarrow aaba$$

$$q_5 \leftrightarrow aabab$$

# Exemple

---

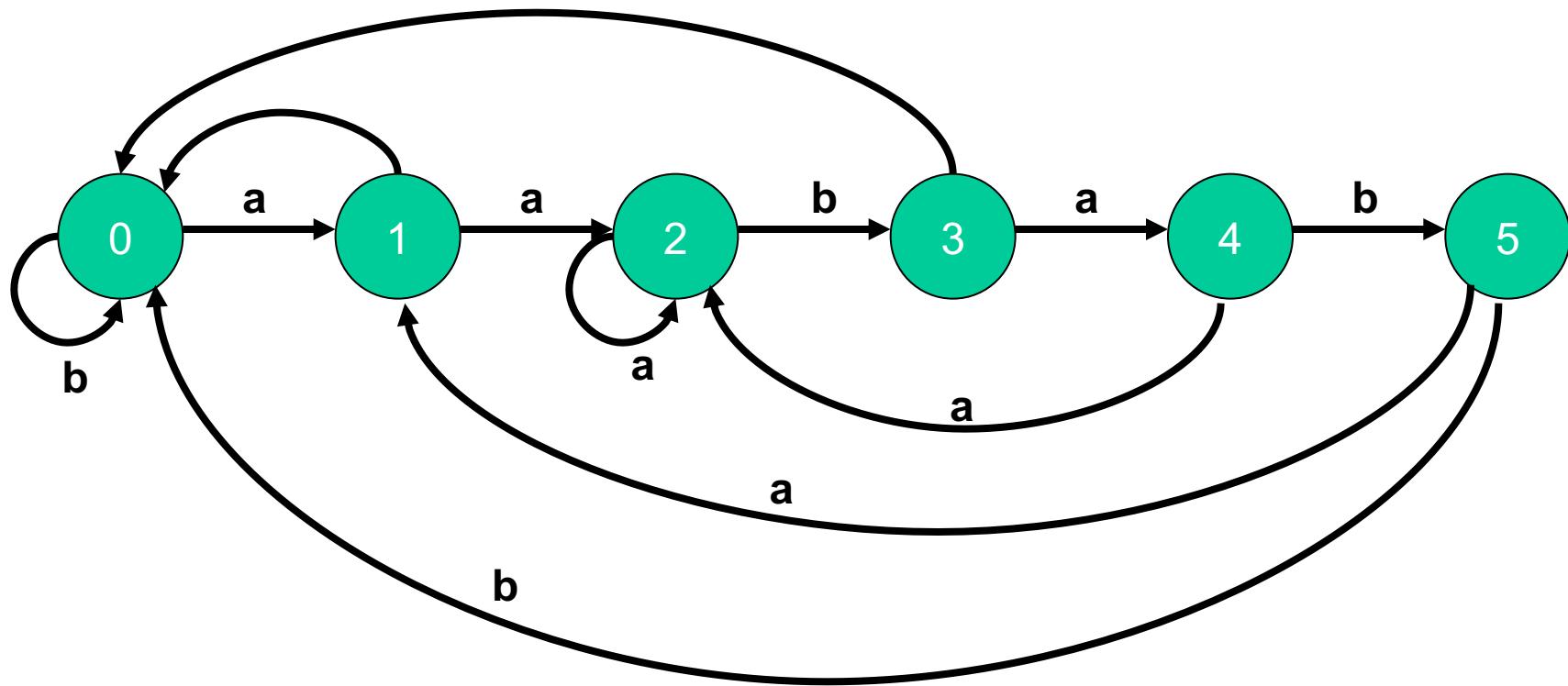
- Fonction de transition  $\delta$ :

	a	b
q0	1	0
q1	2	0
q2	2	3
q3	4	0
q4	2	5
q5	1	0

Patron: aabab

# Exemple

---



# Construction

---

Idée: chaque état a reconnu le préfixe de la chaîne à rechercher qui lui est associe

$$\begin{aligned} q_0 &\leftrightarrow \epsilon \\ q_1 &\leftrightarrow a \\ q_2 &\leftrightarrow aa \\ q_3 &\leftrightarrow aab \\ q_4 &\leftrightarrow aaba \\ q_5 &\leftrightarrow aabab \end{aligned}$$

# Exemple de construction

---

Position: 012345

Chaine: aabab

$$P_0 = \epsilon$$

$$P_1 = a$$

$$P_2 = aa$$

$$P_3 = aab$$

$$P_4 = aaba$$

$$P_5 = aabab$$

$P_i$  est le préfixe de  $P$  de  $i$  lettres

# Exemple de construction

Algorithme

Initialiser  $\delta$  à 0

Pour  $q = 0 : m$

Pour chaque caractère  $a$  dans  $\Sigma_P$

$k = \min(m+1, q+2)$

Répéter

$k = k - 1$

Tant que  $k > 0$  et ( $(P_k \supset P_q a)$  est faux)

$\delta(q, a) = k$

Fin Pour

Fin Pour

Patron: aabab

	a	b
q0	0	0
q1	0	0
q2	0	0
q3	0	0
q4	0	0
q5	0	0

# Transition 0, a

- 1:  $m \leftarrow 5$
- 2:  $q \leftarrow 0$
- 3:  $a \leftarrow 'a'$
- 4:  $k \leftarrow \min(6, 2)$   
 $k \leftarrow 2$
- 5:  $k \leftarrow 1$
- 6:  $a \supset \varepsilon a$  ?  
 $a \supset a$  ✓
- 7:  $\delta(0, a) \leftarrow 1$

Patron: aabab

	a	b
q0	0	0
q1	0	0
q2	0	0
q3	0	0
q4	0	0
q5	0	0

# Transition 0, a

- 1:  $m \leftarrow 5$
- 2:  $q \leftarrow 0$
- 3:  $a \leftarrow 'a'$
- 4:  $k \leftarrow \min(6, 2)$   
 $k \leftarrow 2$
- 5:  $k \leftarrow 1$
- 6:  $a \supset \varepsilon a$  ?  
 $a \supset a$  ✓
- 7:  $\delta(0, a) \leftarrow 1$

Patron: aabab

	a	b
q0	1	0
q1	0	0
q2	0	0
q3	0	0
q4	0	0
q5	0	0

# Transition 0, b

Patron: aabab

- 3:  $a \leftarrow 'b'$
- 4:  $k \leftarrow \min(6, 2)$   
 $\quad \quad \quad \leftarrow 2$
- 5:  $k \leftarrow 1$
- 6:  $a \supset \varepsilon b$   
 $a \supset b$   
 $\quad \quad \quad \leftarrow F$
- 5:  $k \leftarrow 0$
- 6:  $\varepsilon \supset \varepsilon b$   
 $\varepsilon \supset b$   
 $\quad \quad \quad \leftarrow T$
- 7:  $\delta(0, b) \leftarrow 0$

	a	b
q0	1	0
q1	0	0
q2	0	0
q3	0	0
q4	0	0
q5	0	0

# Transition 5, b

---

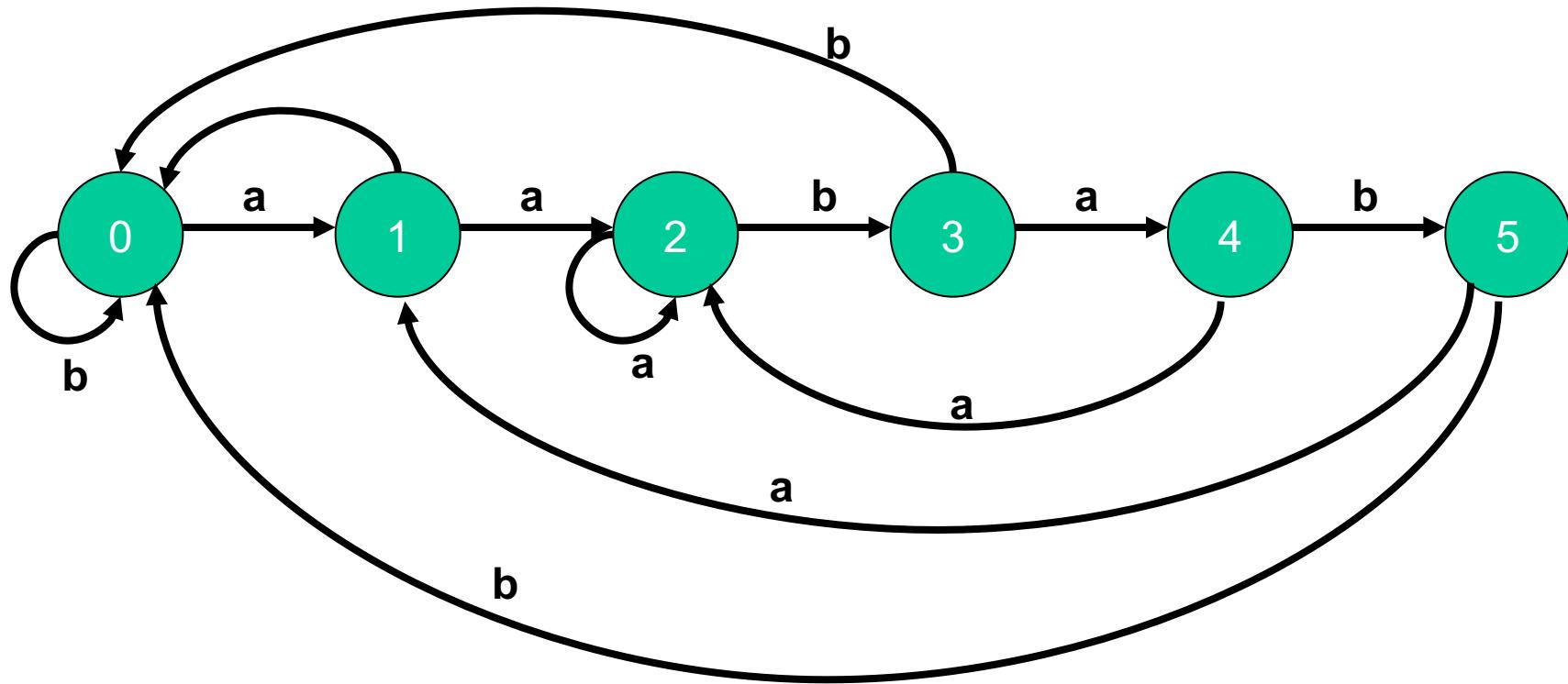
Patron: aabab

	a	b
q0	1	0
q1	2	0
q2	2	3
q3	4	0
q4	2	5
q5	1	0

# Transition 5, b

---

$$\delta(5, b) \leftarrow 0$$



# Automate FSM

---

Dans le meilleur des cas, l'automate FSM donne une complexité

$$O(n) = \Theta(n)$$

La construction de la machine à états peut être coûteuse:  $O(m^3d)$ . Ce qui peut devenir handicapant pour certains problèmes de recherche:

$$O(m^3d_p) + \Theta(n) \text{ vs. } O(m(n-m+1)+m)$$
$$d_p = |\Sigma_p|$$

# Plan

---

Recherche de patron

Problématique

Rabin Karp

Automate FSM

**PLSC**

Problématique

Solution par programmation dynamique

# Problématique

---

## Problématique:

- Étant données deux chaînes de caractères, trouver la plus longue sous-séquence commune.

## Exemples:

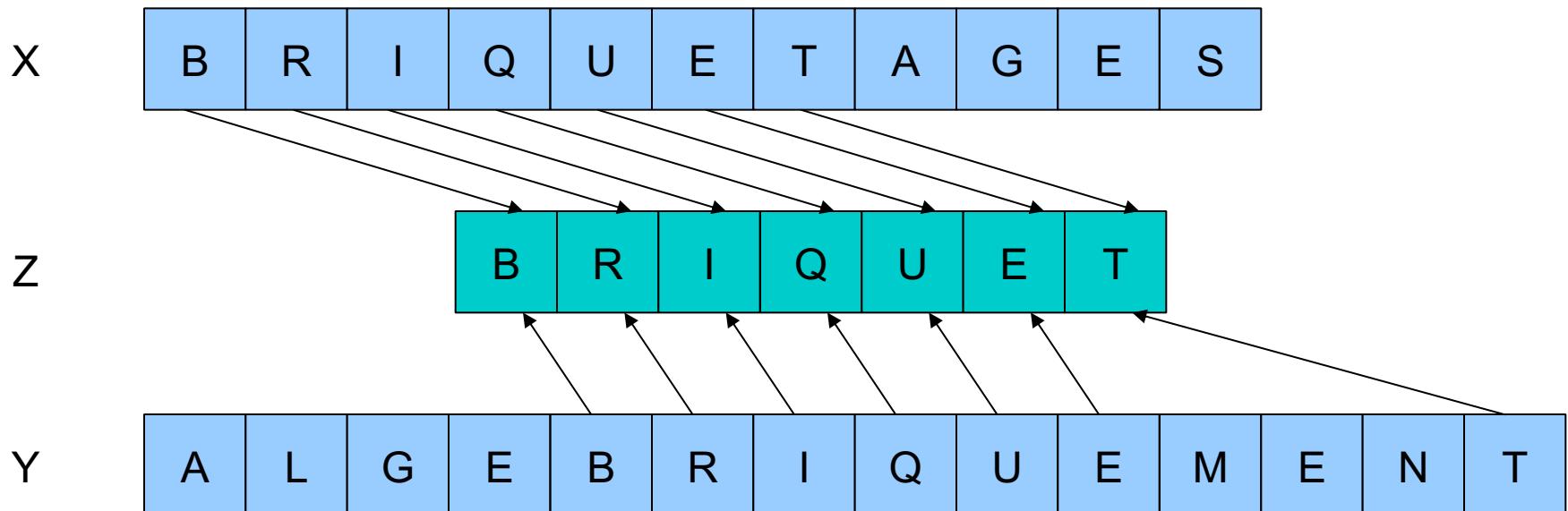
- Comparaison des séquences génétiques ACGT.
- Trouver des modifications dans deux fichiers dans une base de données.

# Problématique

---

## Approche:

La sous-séquence n'est pas continue dans les chaînes d'entrée

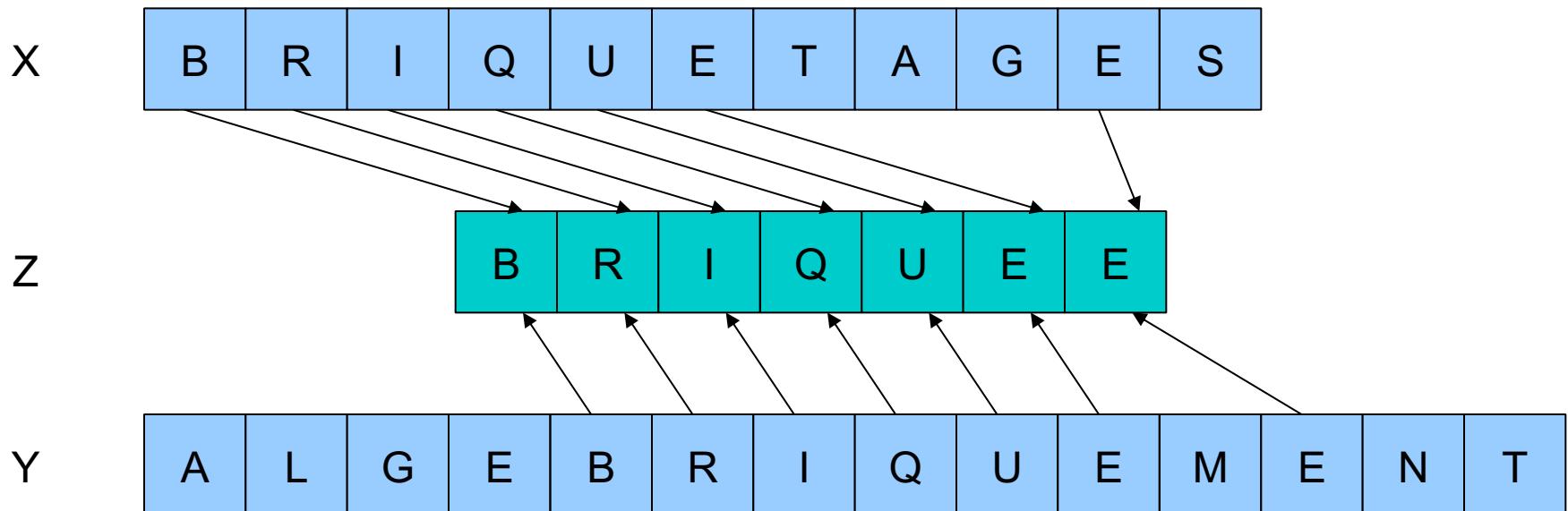


# Problématique

---

## Approche:

La solution n'est pas unique



# Approche naïve

---

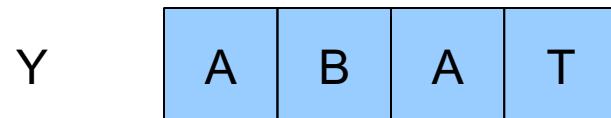
- Il serait possible d'énumérer toutes les sous-séquences de X et Y et de trouver la plus longue sous-séquence commune:



# Approche naïve

---

- Exemple:



1110	ABB	0110	BB
1101	<b>ABA</b>	0101	BA
1011	ABA	0011	BA
0111	BBA	1000	A
1100	AB	0100	B
1010	AB	0010	B
1001	AA	0001	A

1110	<b>ABA</b>	0110	BA
1101	ABT	0101	BT
1011	AAT	0011	AT
0111	BAT	1000	A
1100	AB	0100	B
1010	AA	0010	A
1001	AT	0001	T

# Approche naïve

- Exemple:

X	A	B	B	A	Y	A	B	A	T
COMPLEXITÉ EXPONENTIELLE									
1110	ABB	0110	BB		1110	ABA	0110	BA	
1101	ABA	0101	BA		1101	ABT	0101	BT	
1011	ABA	0011	BA		1011	AAT	0011	AT	
0111	BBA	1000	A		0111	BAT	1000	A	
1100	AB	0100	B		1100	AB	0100	B	
1010	AB	0010	B		1010	AA	0010	A	
1001	AA	0001	A		1001	AT	0001	T	

# Plan

---

Recherche de patron

Problématique

Rabin Karp

Automate FSM

PLSC

Problématique

Solution par programmation dynamique

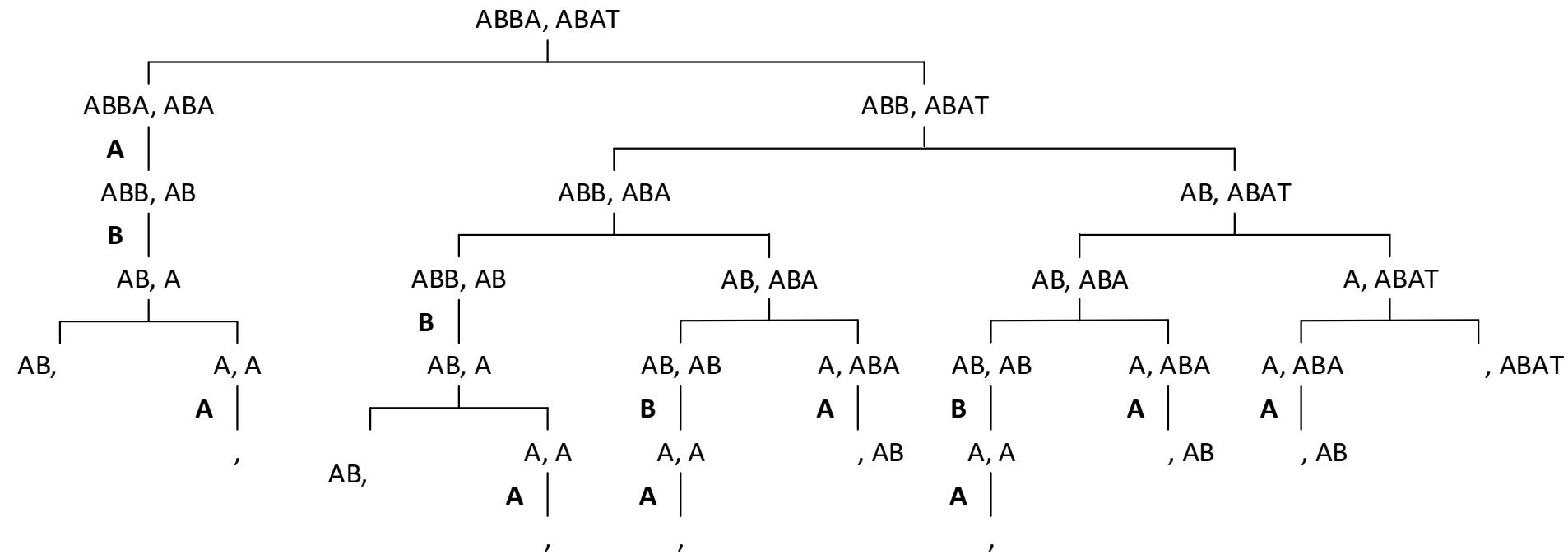
# Formulation récursive

---

- Pour deux séquences d'entrée  $X[1..n]$ ,  $Y[1..m]$ , et une PLSC  $Z[1..k]$  de  $X$  et  $Y$ , on note:
  - Si  $X[n] = Y[m]$ ,
    - alors  $Z[k]=X[n]$  et  $Z[1..k-1]$  est PLSC de  $X[1..n-1]$   $Y[1..m-1]$
  - Si  $X[n] \neq Y[m]$ ,
    - alors si  $Z[k]\neq X[n]$ ,  $Z[1..k]$  est PLSC de  $X[1..n-1]$   $Y[1..m]$
    - Si  $X[n] \neq Y[m]$ ,
      - alors si  $Z[k]\neq Y[m]$ ,  $Z[1..k]$  est PLSC de  $X[1..n]$   $Y[1..m-1]$

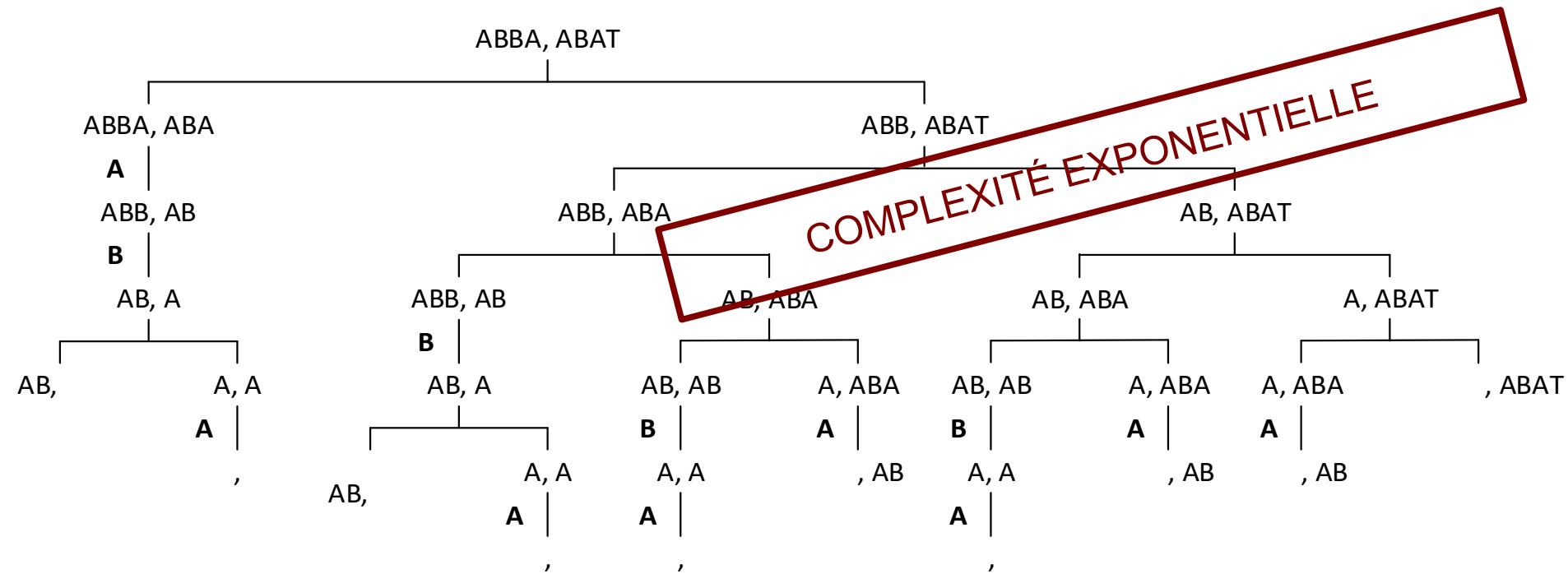
# Solution par programmation dynamique

La formulation récursive donne l'arbre d'exécution suivant:



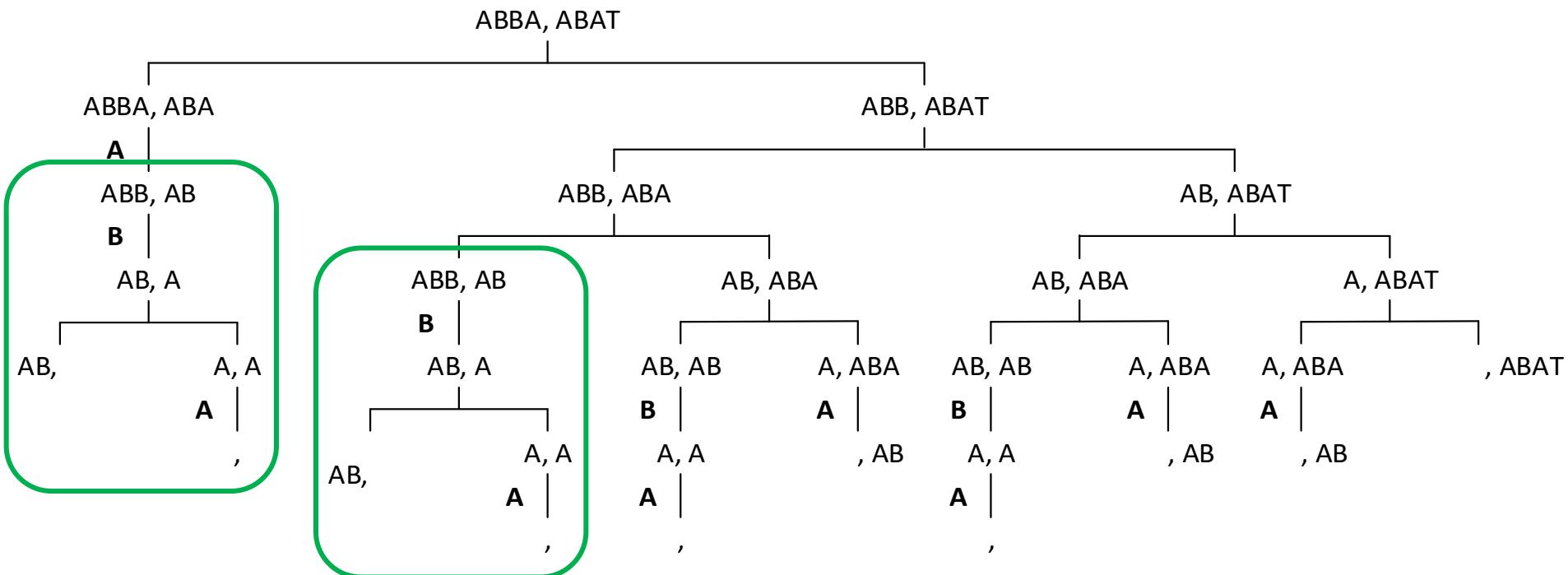
# Solution par programmation dynamique

La formulation récursive donne l'arbre d'exécution suivant:



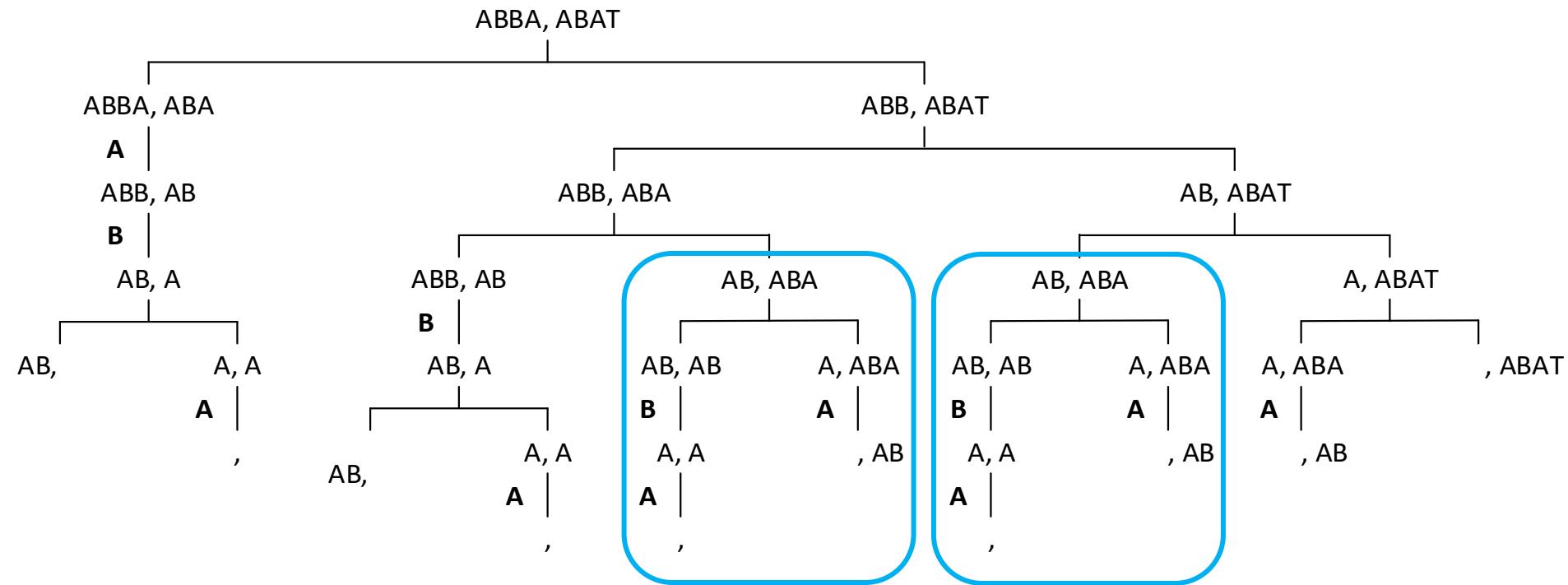
# Solution par programmation dynamique

La formulation récursive donne l'arbre d'exécution suivant:



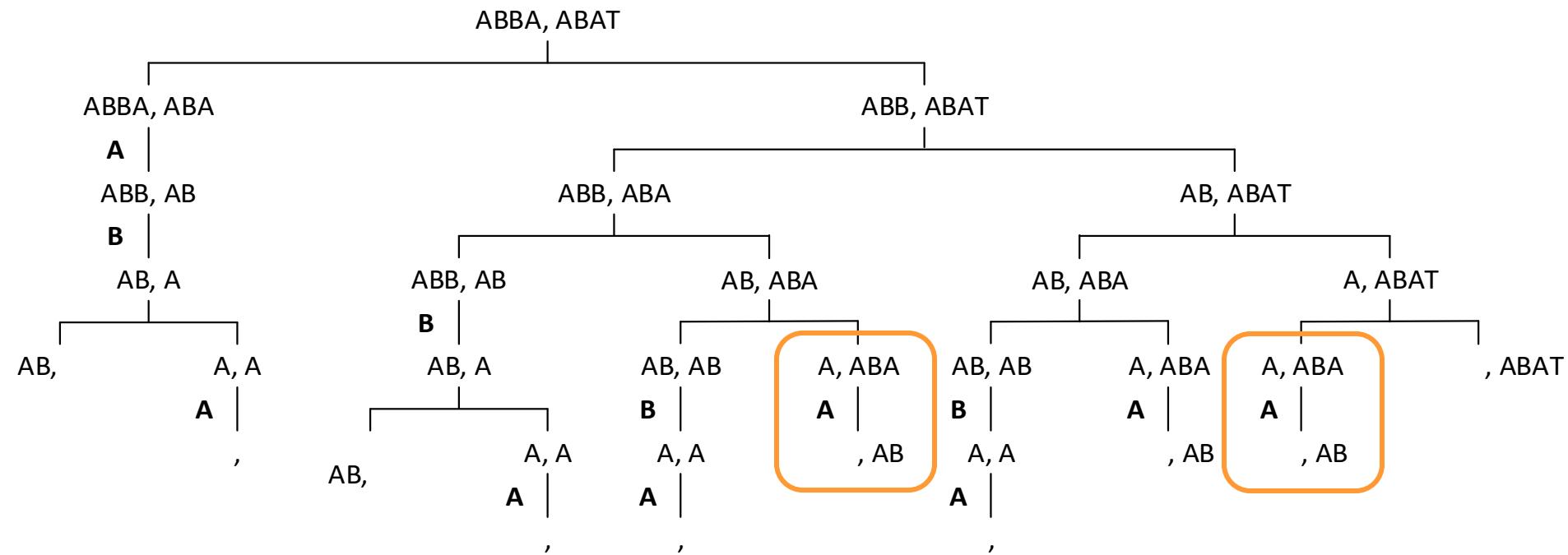
# Solution par programmation dynamique

La formulation récursive donne l'arbre d'exécution suivant:



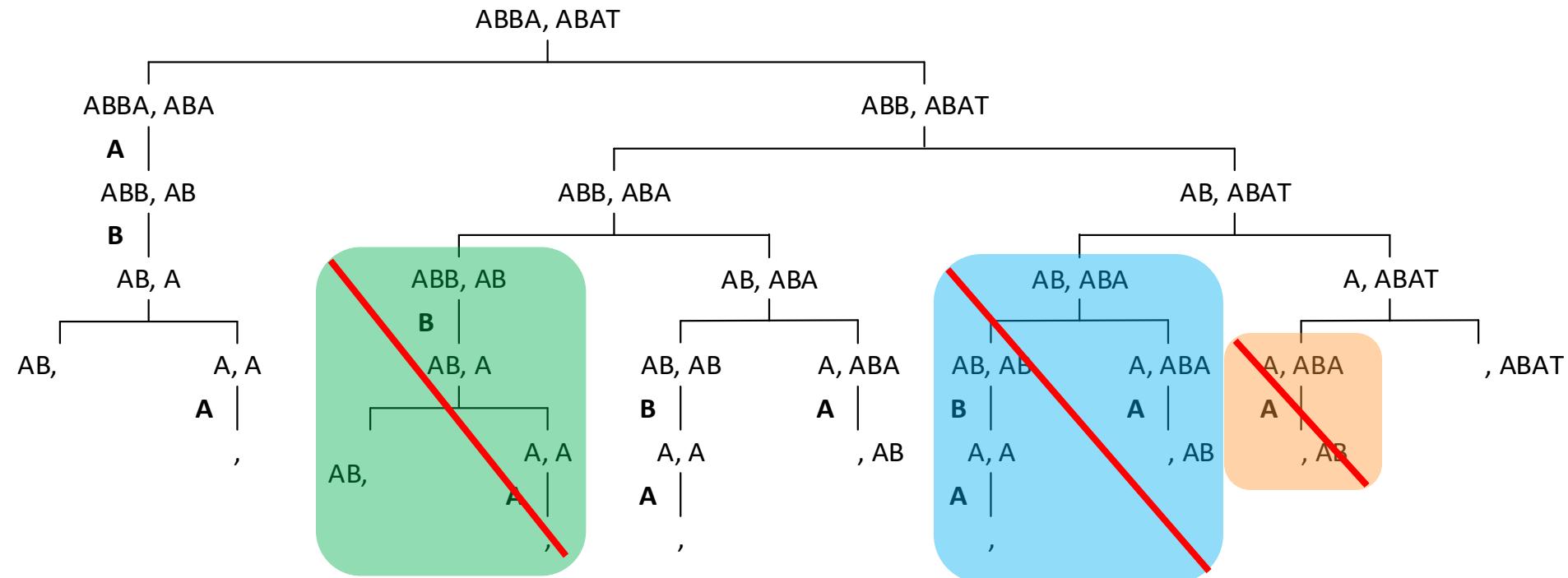
# Solution par programmation dynamique

La formulation récursive donne l'arbre d'exécution suivant:



# Solution par programmation dynamique

La solution par programmation dynamique consiste à construire l'arbre de bas vers le haut, et de réutiliser les résultats redondants:



# Solution par programmation dynamique

---

- Il est alors possible de formuler un algorithme de **programmation dynamique**:
  - On définit deux tables 2-D auxiliaires  $d$  (direction) et  $t$  (taille)
  - $t$  a une taille  $(n+1) \times (m+1)$  (initialisée à 0)
  - $d$  a une taille  $n \times m$
  - On enregistre dans  $t[i+1, j+1]$  la taille de la PLSC de  $X[1..i], Y[1..j]$
  - On enregistre dans  $d[i, j]$  la direction (HAUT, GAUCHE, DIAG) vers l'origine pour trouver la PLSC de  $X[1..i], Y[1..j]$

# Solution par programmation dynamique

---

**Pour**  $i = 1 : n$

**Pour**  $j = 1 : m$

**Si**  $X[i] == Y[j]$

$$t[i+1, j+1] = t[i, j] + 1$$

$$d[i, j] = \text{DIAG}(\nwarrow)$$

**Sinon Si**  $t[i, j+1] \geq t[i+1, j]$

$$t[i+1, j+1] = t[i, j+1]$$

$$d[i, j] = \text{HAUT}(\uparrow)$$

**Sinon**

$$t[i+1, j+1] = t[i+1, j]$$

$$d[i, j] = \text{GAUCHE}(\leftarrow)$$

# Solution par programmation dynamique

---

$d$	A	B	A	T
A				
B				
B				
A				

$t$		A	B	A	T
	0	0	0	0	0
A	0				
B	0				
B	0				
A	0				

# Solution par programmation dynamique

---

$d$	A	B	A	T
A				
B				
B				
A				

$t$		A	B	A	T
	0	0	0	0	0
A	0	1			
B	0				
B	0				
A	0				

# Solution par programmation dynamique

---

$d$	A	B	A	T
A				
B				
B				
A				

$t$		A	B	A	T
	0	0	0	0	0
A	0	1			
B	0				
B	0				
A	0				

# Solution par programmation dynamique

---

$d$	A	B	A	T
A				
B				
B				
A				

$t$		A	B	A	T
	0	0	0	0	0
A	0	1			
B	0				
B	0				
A	0				

# Solution par programmation dynamique

---

$d$	A	B	A	T
A		←		
B				
B				
A				

$t$		A	B	A	T
	0	0	0	0	0
A	0	1	1		
B	0				
B	0				
A	0				

# Solution par programmation dynamique

---

$d$	A	B	A	T
A				
B				
B				
A				

$t$		A	B	A	T
	0	0	0	0	0
A	0	1	1		
B	0				
B	0				
A	0				

# Solution par programmation dynamique

---

$d$	A	B	A	T
A				
B				
B				
A				

$t$		A	B	A	T
	0	0	0	0	0
A	0	1	1	1	
B	0				
B	0				
A	0				

# Solution par programmation dynamique

---

$d$	A	B	A	T
A				
B				
B				
A				



$t$		A	B	A	T
	0	0	0	0	0
A	0	1	1	1	
B	0				
B	0				
A	0				

# Solution par programmation dynamique

---

$d$	A	B	A	T
A				
B				
B				
A				

Arrows indicate transitions from the A column to the T column:

- An arrow from the first row's A cell to the first row's T cell.
- An arrow from the second row's A cell to the first row's T cell.
- An arrow from the third row's A cell to the first row's T cell.
- An arrow from the fourth row's A cell to the first row's T cell.

$t$		A	B	A	T
	0	0	0	0	0
A	0	1	1	1	
B	0				
B	0				
A	0				

# Solution par programmation dynamique

---

$d$	A	B	A	T
A				
B				
B				
A				

Arrows indicate dependencies: from T to A, from T to B, from A to A, and from B to A.

$t$		A	B	A	T
	0	0	0	0	0
A	0	1	1	1	1
B	0				
B	0				
A	0				

# Solution par programmation dynamique

---

$d$	A	B	A	T
A				
B				
B				
A				

Arrows indicate transitions between states:

- From A to A
- From A to T
- From B to A
- From B to T
- From B to B

$t$		A	B	A	T
	0	0	0	0	0
A	0	1	1	1	1
B	0	1	2	2	2
B	0				
A	0				

# Solution par programmation dynamique

---

$d$	A	B	A	T
A				
B				
B				
A				

Arrows indicate transitions between states:

- From A to B
- From B to A
- From A to T
- From B to T
- From A to A (diagonal)
- From B to B (diagonal)

$t$		A	B	A	T
	0	0	0	0	0
A	0	1	1	1	1
B	0	1	2	2	2
B	0	1	2	2	2
A	0				

# Solution par programmation dynamique

---

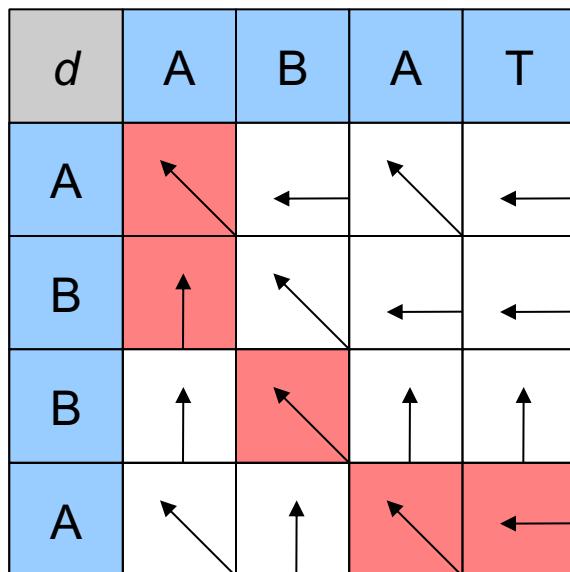
$d$	A	B	A	T
A				
B				
B				
A				

A 5x5 grid with columns labeled d, A, B, A, T. The first column (d) is shaded grey. The last column (T) is shaded green. Arrows indicate transitions between states: A to B, B to A, A to T, B to T, and T to A.

$t$		A	B	A	T
	0	0	0	0	0
A	0	1	1	1	1
B	0	1	2	2	2
B	0	1	2	2	2
A	0	1	2	3	3

# Solution par programmation dynamique

---



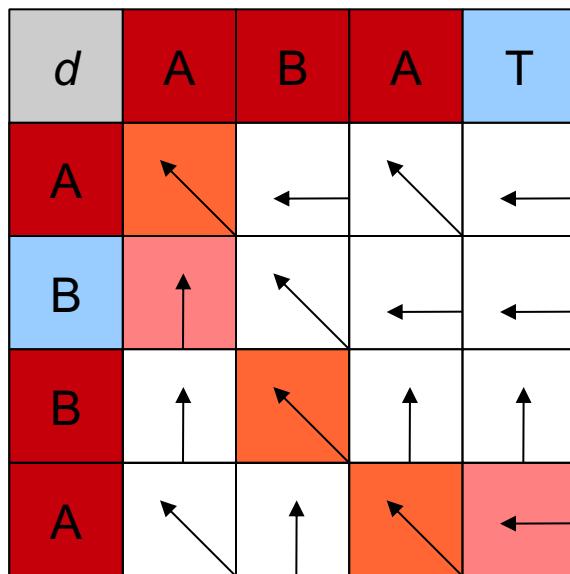
$t$		A	B	A	T
	0	0	0	0	0
A	0	1	1	1	1
B	0	1	2	2	2
B	0	1	2	2	2
A	0	1	2	3	3

En suivant la direction des flèches, on peut retrouver la PLSC

---

# Solution par programmation dynamique

---

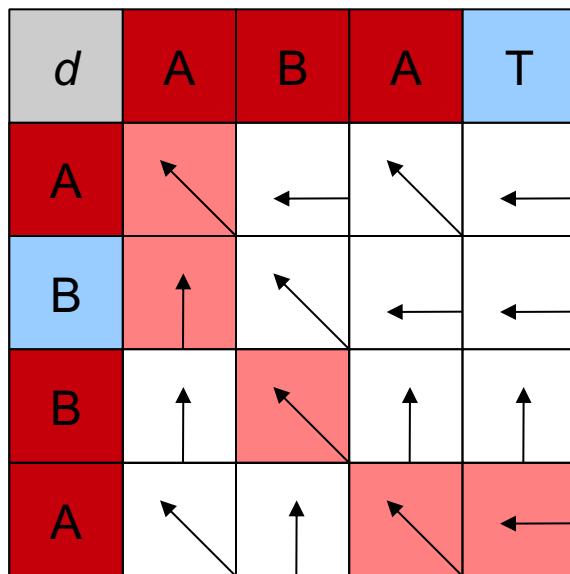


<i>t</i>		A	B	A	T
	0	0	0	0	0
A	0	1	1	1	1
B	0	1	2	2	2
B	0	1	2	2	2
A	0	1	2	3	3

Les caractères de la PLSC sont aux flèches diagonales

---

# Solution par programmation dynamique



$t$		A	B	A	T
	0	0	0	0	0
A	0	1	1	1	1
B	0	1	2	2	2
B	0	1	2	2	2
A	0	1	2	3	3

On pourrait minimiser la mémoire de  $t$  à deux lignes

# Solution par programmation dynamique

---

- Appliquer l'algorithme pour l'exemple suivant:

X      

O	V	A	L	E	S
---	---	---	---	---	---

Z      

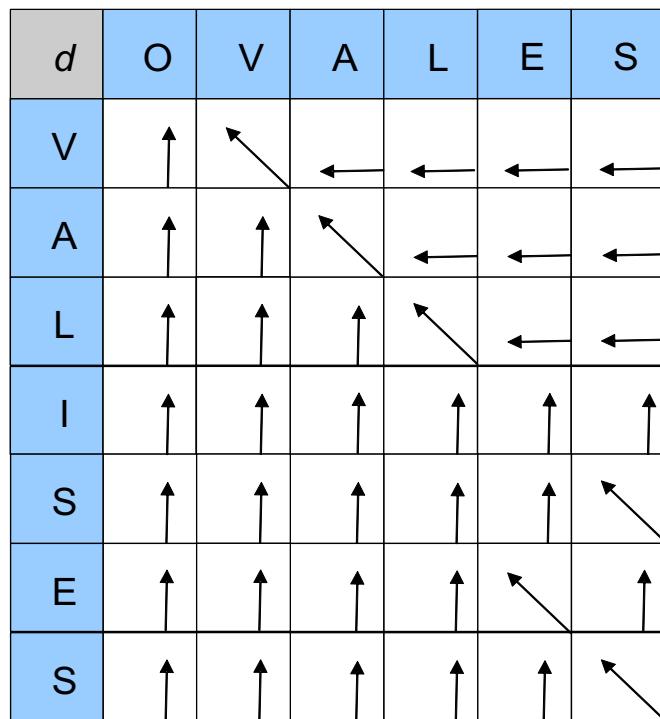
?
---

Y      

V	A	L	I	S	E	S
---	---	---	---	---	---	---

# Solution par programmation dynamique

---



<i>t</i>		O	V	A	L	E	S
	0	0	0	0	0	0	0
V	0	0	1	1	1	1	1
A	0	0	1	2	2	2	2
L	0	0	1	2	3	3	3
I	0	0	1	2	3	3	3
S	0	0	1	2	3	3	4
E	0	0	1	2	3	4	4
S	0	0	1	2	3	4	5