

# Programmation dynamique

# Paradigmes algorithmiques

**Algorithme glouton** : construit une solution de manière incrémentale, en optimisant un critère de manière locale.

**Diviser pour régner** : divise un problème en sous-problèmes **indépendants**, résout chaque sous-problème, et combine les solutions des sous-problèmes pour former une solution du problème initial.

**Programmation dynamique** : divise un problème en sous-problèmes qui sont non indépendants, et cherche (et stocke) des solutions de sous-problèmes de plus en plus grands.

# Programmation dynamique



**Historique** : paradigme développé par Richard Bellman en 1953.

**Technique de conception d'algorithmes très générale et performante.**

Permet de résoudre de nombreux **problèmes d'optimisation**.

**Exemples d'algorithmes** de programmation dynamique :  
alignement de séquences ADN, algorithme de plus courts chemins (Bellman-Ford), commande Unix diff, etc.

“An interesting question is, ‘Where did the name dynamic programming, come from?’ The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I’m not using the term lightly; I’m using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? (...) I decided therefore to use the word, ‘programming.’ I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let’s kill two birds with one stone. Let’s take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it’s impossible to use the word dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It’s impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities”

**Bellman**

# Programmation dynamique

Idée : ``recherche exhaustive intelligente``

(sous-problèmes + réutilisation de solutions déjà calculées).

Exemple : calcul de  $F_n$  :  $n^{\text{ème}}$  nombre de Fibonacci

$$\begin{cases} F_1 = F_2 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

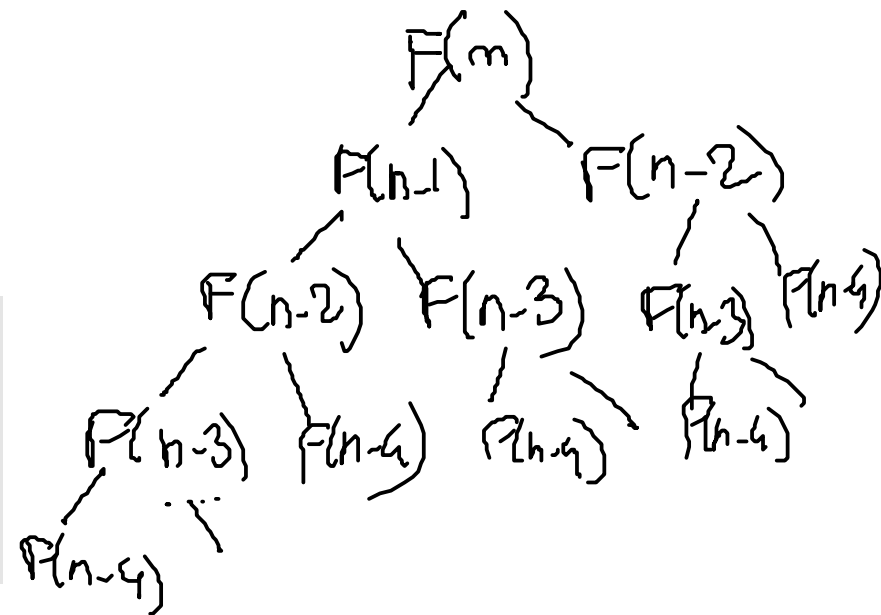
Algorithme naïf :

Fib(n) :

si  $n \leq 2$  alors  $F = 1$

sinon  $F = \text{Fib}(n-1) + \text{Fib}(n-2)$

retourner  $F$



Complexité exponentielle :  $T(n) = T(n-1) + T(n-2) + \Theta(1) \approx \varphi^n$

# Mémoïsation

Exemple :

```
mémo : tableau de n entiers  
est-dans-mémo : tableau de n booléens initialisés à faux  
Fib(n) :  
    si est-dans-mémo[n] retourner mémo[n]  
    si  $n \leq 2$  alors  $F = 1$   
    sinon  $F = \text{Fib}(n-1) + \text{Fib}(n-2)$   
    mémo[n] = F ; est-dans-mémo[n] = vrai  
    retourner F
```

Fib(k) induit des appels récursifs seulement la première fois qu'elle est appelée.

Ceci peut être fait pour tout algorithme récursif.

Mémoïser = *conserver à la fin de l'exécution d'une fonction le résultat associé aux arguments d'appels, pour ne pas avoir à recalculer ce résultat lors d'un autre appel récursif.*

# Mémoïsation : complexité

## Exemple :

**mémo** : tableau de n entiers  
**est-dans-mémo** : tableau de n booléens initialisés à faux  
Fib(n) :  
    **si est-dans-mémo[n] retourner mémo[n]**  
    si  $n \leq 2$  alors  $F = 1$   
    sinon  $F = \text{Fib}(n-1) + \text{Fib}(n-2)$   
    **mémo[n] = F ; est-dans-mémo[n] = vrai**  
    retourner F

Nombre d'appels non ``mémoïsés" : n

Coût d'un appel (sans compter les appels récursifs ``non mémoïsés") :  $\Theta(1)$   
(mémo[i] est retourné en  $\Theta(1)$  si mémo est un tableau)

Complexité temporelle :  $\Theta(n)$

# Programmation dynamique

Idée : mémoriser et réutiliser les solutions de sous-problèmes qui aident à résoudre le problème.

Complexité temporelle =  
nombre de sous-problèmes  $\times$  (complexité par sous-problème\*)

\* On ne compte pas les appels récursifs.



# Programmation dynamique (2)

Deuxième manière de voir la programmation dynamique :

approche "du bas vers le haut"

Fib

1	2	3	4	5	6
1	1	2	3	5	8

Exemple :

```
Fib = []
```

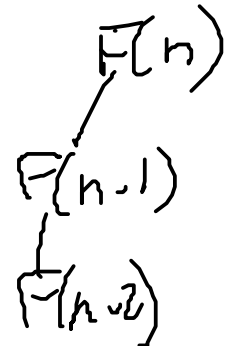
```
pour k de 1 à n :
```

```
    si  $k \leq 2$  alors  $F = 1$ 
```

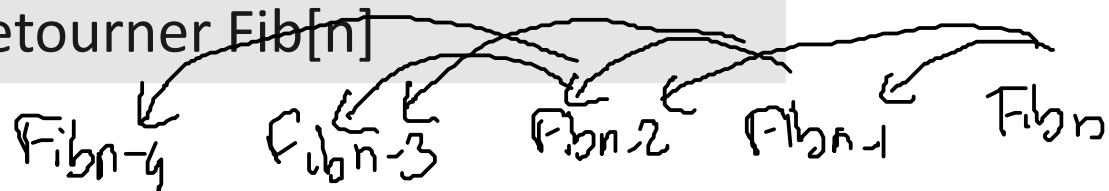
```
    sinon  $F = \text{Fib}[k-1] + \text{Fib}[k-2]$ 
```

```
     $\text{Fib}[k] = F$ 
```

```
retourner  $\text{Fib}[n]$ 
```



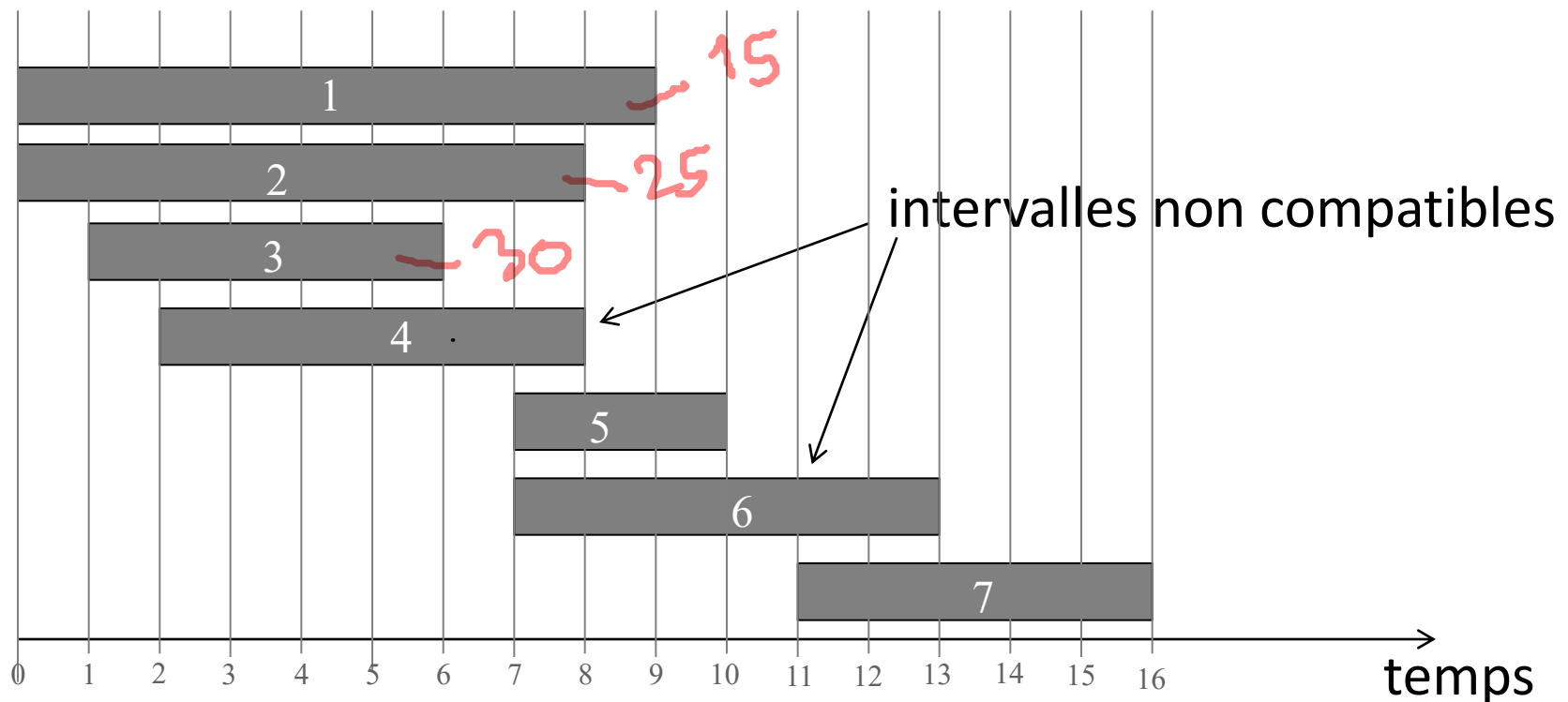
Cas général :



- Exactly les **mêmes calculs** que dans la **version mémorisée**.
- Les **sous-problèmes sont traités dans un ordre topologique**.
- Complexité temporelle : évidente
- Permet souvent de baisser la complexité spatiale.

# Exemple : ordonnancement d'intervalles pondérés

- L'intervalle  $i$  commence en  $d_i$ , se termine en  $f_i$  et a une valeur  $v_i$ .
- Deux intervalles sont **compatibles** s'ils ne s'intersectent pas.
- **But** : déterminer un **sous-ensemble** d'intervalles mutuellement compatibles **de valeur maximale**.



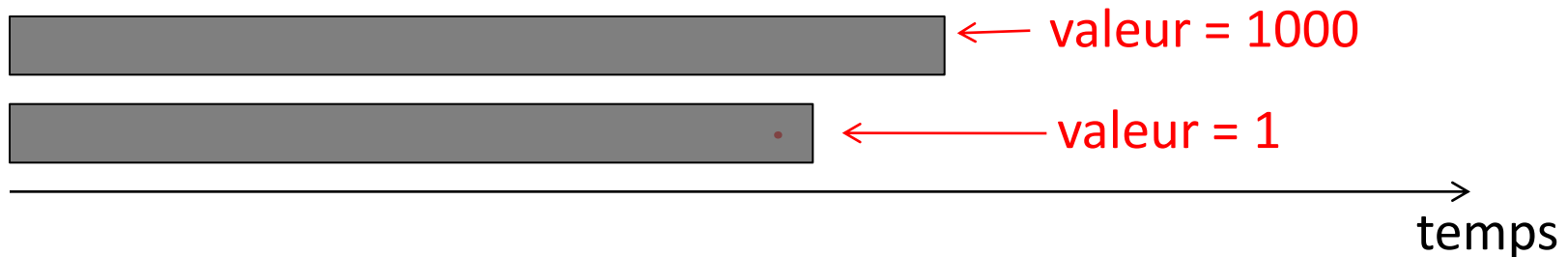
# Ordonnancement d'intervalles pondérés

**Rappel** : Algorithme “celui qui termine le plus tôt d'abord”

- Considérer les intervalles dans l'ordre de leurs dates de fins croissantes.
- Ajouter un intervalle au sous-ensemble des intervalles choisis s'il est compatible avec ces intervalles.

Cet algorithme est optimal si tous les poids sont égaux à 1.

Il peut être mauvais dans la version pondérée :



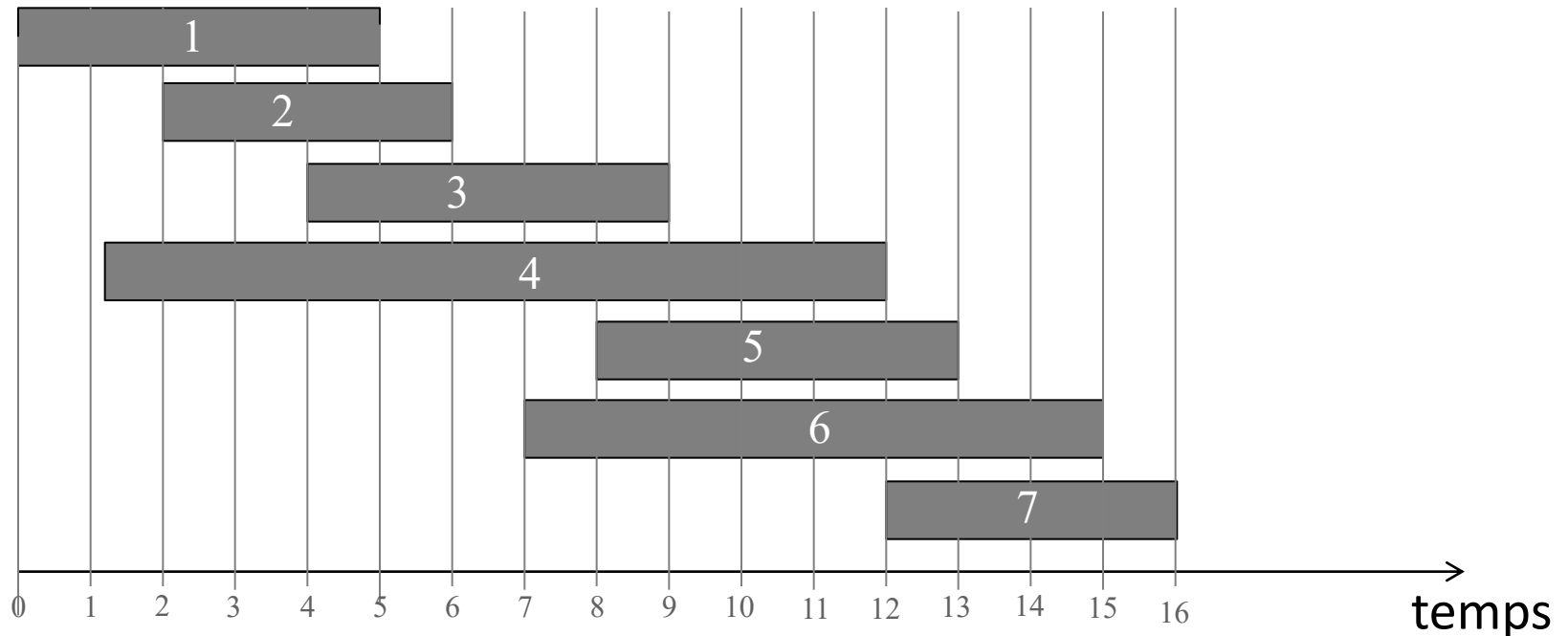
# Ordonnancement d'intervalles pondérés

**Notation** : on numérote les intervalles par dates de fin croissantes :

$$f_1 \leq f_2 \leq \dots \leq f_n$$

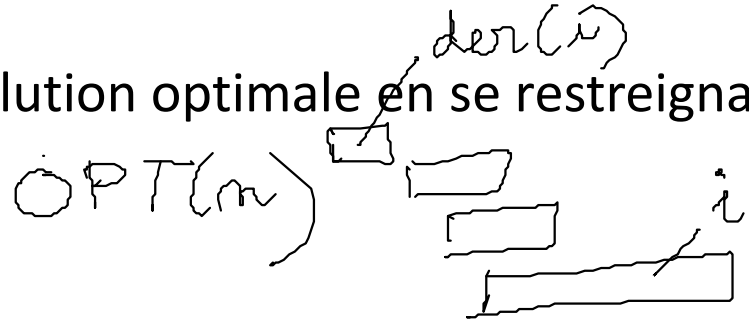
**Définition** :  $\text{der}(j)$  = plus grand numéro  $i < j$  tel que l'intervalle  $i$  est compatible avec l'intervalle  $j$ .

**Exemple** :  $\text{der}(7)=4$  ;  $\text{der}(6)= 2$  ;  $\text{der}(2)=0$ .



# Ordonnancement d'intervalles pondérés : choix binaire

**Notation** :  $OPT(i)$  = valeur d'une solution optimale en se restreignant aux intervalles 1, ..., i.



**Cas 1 : i est choisi dans OPT**

- On obtient la valeur de i :  $v_i$
- On ne peut pas choisir les intervalles  $\{der(i)+1, der(i)+2, \dots, i-1\}$
- On doit choisir la solution optimale du problème restreint aux intervalles 1, 2, ...,  $der(i)$   $v_i + OPT(der(i))$ .

**Cas 2 : i n'est pas dans OPT**  $OPT(i) = OPT(i-1)$

- On doit choisir la solution optimale du problème restreint aux intervalles 1, 2, ...,  $i-1$

$$OPT(i) = \begin{cases} 0 & \text{si } i=0 \\ \max \{ v_i + OPT(der(i)) , OPT(i-1) \} & \text{sinon} \end{cases}$$

# Ordonnancement d'intervalles pondérés : algorithme naïf

Entrée :  $n, d[1..n], f[1..n], v[1..n]$

Trier les intervalles de façon à ce que  $f[1] \leq f[2] \leq \dots \leq f[n]$ .

Calculer  $der[1], der[2], \dots, der[n]$

Calcule\_OPT( $i$ )

si  $i=0$  alors  $s = 0$

sinon  $s = \max \{v[i] + \text{Calcule\_OPT}(der[i]) , \text{Calcule\_OPT}(i-1) \}$

retourner  $s$

# Quelle est la complexité de cet algorithme ?

Entrée :  $n, d[1..n], f[1..n], v[1..n]$

Trier les intervalles de façon à ce que  $f[1] \leq f[2] \leq \dots \leq f[n]$ .

Calculer  $der[1], der[2], \dots, der[n]$

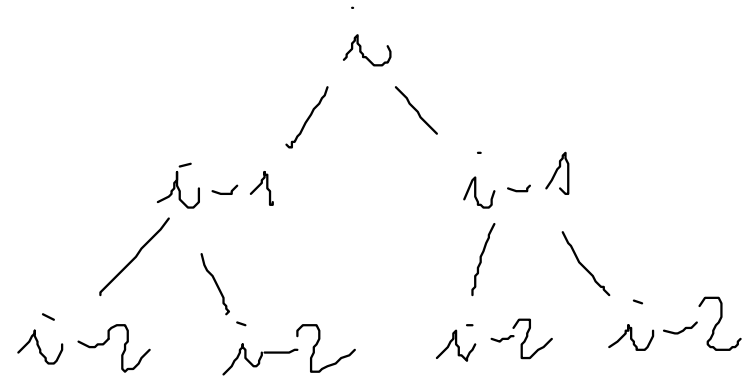
Calcule\_OPT( $i$ )

si  $i=0$  alors  $s = 0$

sinon  $s = \max \{v[i] + \text{Calcule\_OPT}(der[i]) , \text{Calcule\_OPT}(i-1) \}$

retourner  $s$

- A.  $O(n)$
- B.  $O(n \log n)$
- C.  $O(n^2)$
- D.  $O(2^n)$



# Ordonnancement d'intervalles pondérés : mémoïsation

Entrée :  $n, d[1..n], f[1..n], v[1..n]$

Trier les intervalles de façon à ce que  $f[1] \leq f[2] \leq \dots \leq f[n]$ .

Calculer  $der[1], der[2], \dots, der[n]$

pour  $j$  allant de 1 à  $n$

    mémo[ $j$ ] = vide

    mémo[0] = 0

Calculer\_OPT( $i$ )

    si mémo[ $i$ ] est vide alors

        mémo[ $i$ ] =  $\max \{v[i] + \text{Calculer\_OPT}(der[i]) , \text{Calculer\_OPT}(i-1) \}$

    retourner mémo[ $i$ ]



# Quelle est la complexité de cet algorithme ?

$O(n \log n)$  {  
Entrée :  $n, d[1..n], f[1..n], v[1..n]$   
Trier les intervalles de façon à ce que  $f[1] \leq f[2] \leq \dots \leq f[n]$ . —  $O(n \log n)$   
Calculer  $der[1], der[2], \dots, der[n]$ , —  $O(n)$   
Initialiser le tableau **memo** à vide ( $memo[0] = 0$ )  $O(n)$   
Calcule\_OPT(i)  
si  $memo[i]$  est vide alors  
     $memo[i] = \max \{v[i] + \text{Calcule\_OPT}(der[i]) , \text{Calcule\_OPT}(i-1) \}$   
retourner  $memo[i]$   
}  
 $O(n)$

A.  $O(n)$

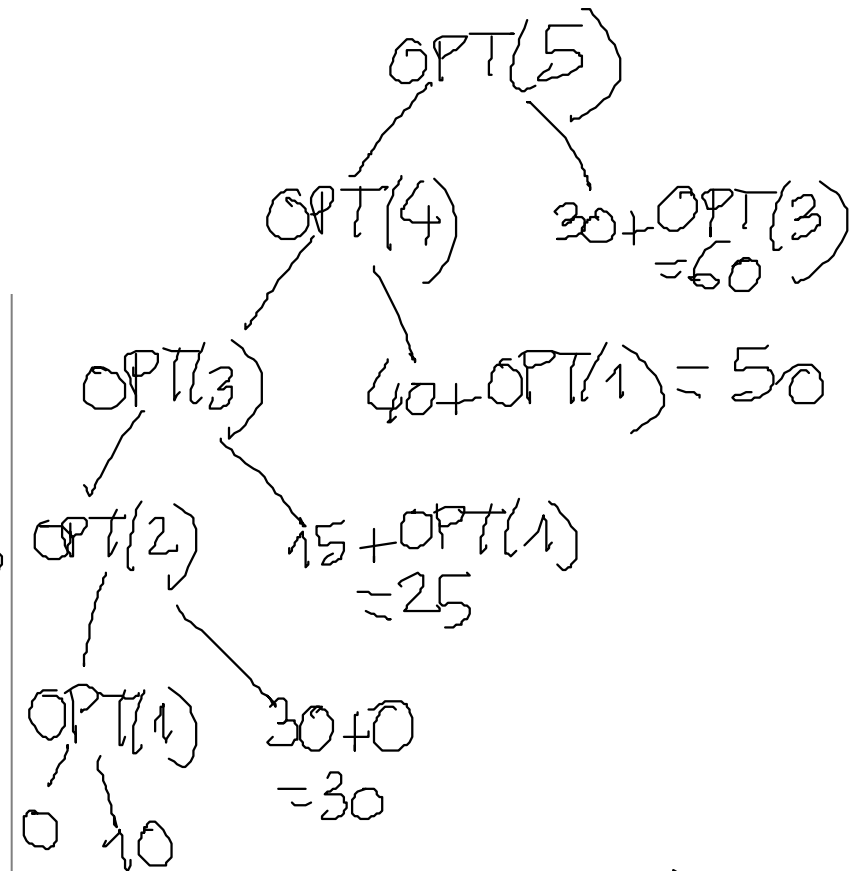
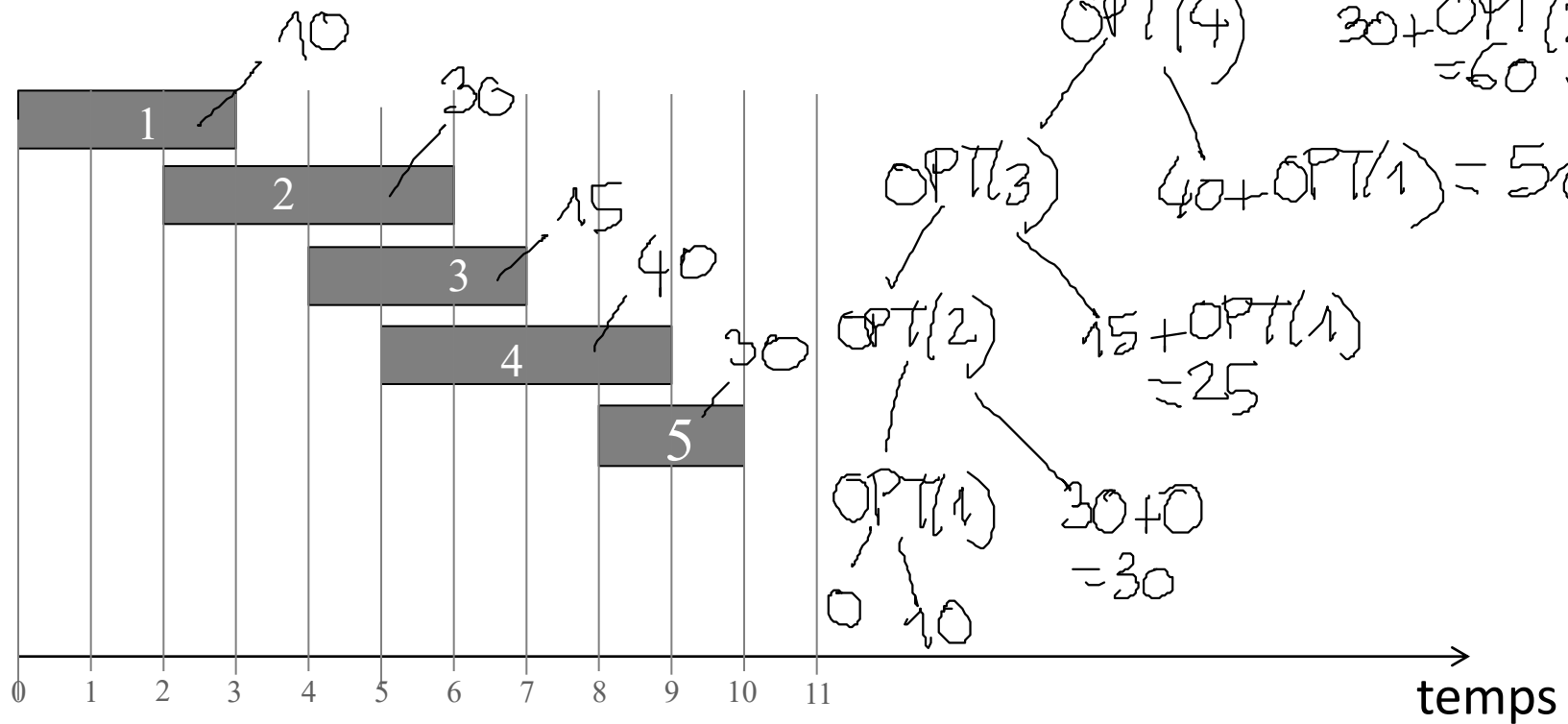
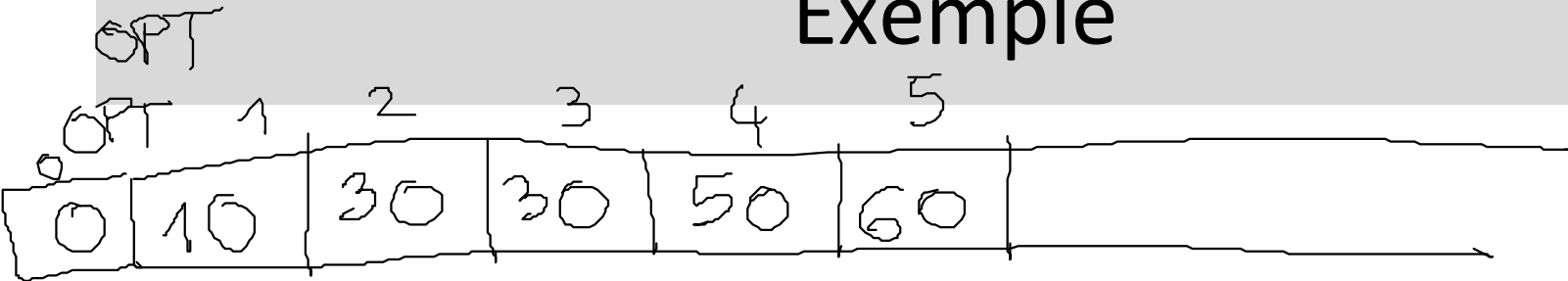
B.  $O(n \log n)$

C.  $O(n^2)$

D.  $O(2^n)$

Complexité : Initialisation :  $\Theta(n \log n)$   
Calcule\_OPT(n) :  $\Theta(n)$

# Exemple



# Ordonnancement d'intervalles pondérés : approche ``du bas vers le haut''

Entrée :  $n, d[1..n], f[1..n], v[1..n]$

Trier les intervalles de façon à ce que  $f[1] \leq f[2] \leq \dots \leq f[n]$ .

Calculer  $der[1], der[2], \dots, der[n]$

mémo[0] = 0

pour  $i$  allant de 1 à  $n$

    mémo[i] = max {  $v[i] + \text{mémo}[der[i]]$  ,  $\text{mémo}[i-1]$  }

Complexité : Initialisation :  $\Theta(n \log n)$

    boucle :  $\Theta(n)$

# Exemple

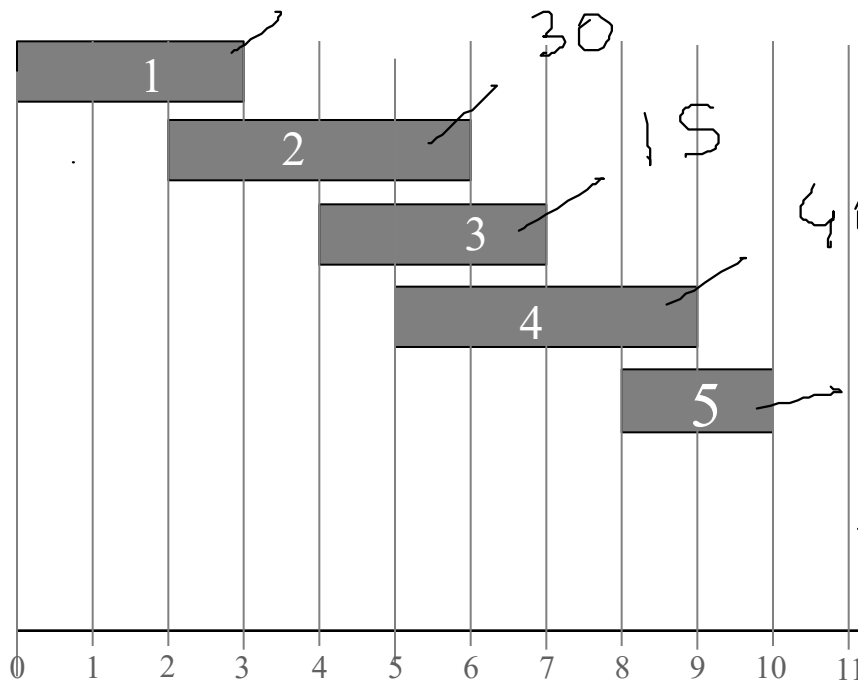
backtrack retour arrière

	<del>1</del>	2	<del>3</del>	<del>4</del>	5	
OPT	10	30	30	50	60	
SL	{1}	{2}	{1,2}	{1,4}	{2,5}	

10

den

1	2	3	4	5
0	0	1	1	3



$OPT(3) = 30$   
 3 pas?  $15 + 10 = 25$   
 3 pas? 30  
 $v_i + OPT(den/i)$

Int pos = {5, 2}

# Ordonnancement d'intervalles pondérés : comment retrouver la solution optimale ?

$m-5$

↙

```
Trouver_solution(i)
  si  $i = 0$  alors
    retourner  $\emptyset$ 
  si  $v[i] + \text{mémo}(\text{der}[i]) > \text{mémo}[i-1]$  alors
    retourner  $\{i\} \cup \text{Trouver\_solution}(\text{der}[i])$ 
  sinon
    retourner Trouver_solution(i-1)
```

Complexité :  $\Theta(n)$       (nombre d'appels récurifs  $\leq n$ )

# Quiz

Soient A et B deux algorithmes de programmation dynamique basés sur la même relation de récurrence. A est un algorithme **récuratif avec mémoïsation**, et B un algorithme **itératif**.

A et B ont-ils la **même complexité** temporelle ?

- A. Oui
- B. Non

# Quiz

**Problème SubsetSum** : soit un entier  $W$  et  $n$  entiers positifs  $S = \{a_1, a_2, a_3, \dots, a_n\}$ . Existe-t-il un sous-ensemble de  $S$  dont la somme des éléments est  $W$ ?

$W=10$   $\{1, 5, 3, 2, 12\}$   
oui  
 $5+3+2=10$

Programme dynamique pour SubsetSum : utilise un tableau de booléens  $X$  à  $n$  lignes et  $W+1$

colonnes t.q.  $X[i, j]$  (avec  $1 \leq i \leq n$ ,  $0 \leq j \leq W$ ) est

**Vrai** ssi il existe un sous-ensemble de  $\{a_1, a_2, \dots, a_i\}$  dont la somme des éléments est  $j$ .

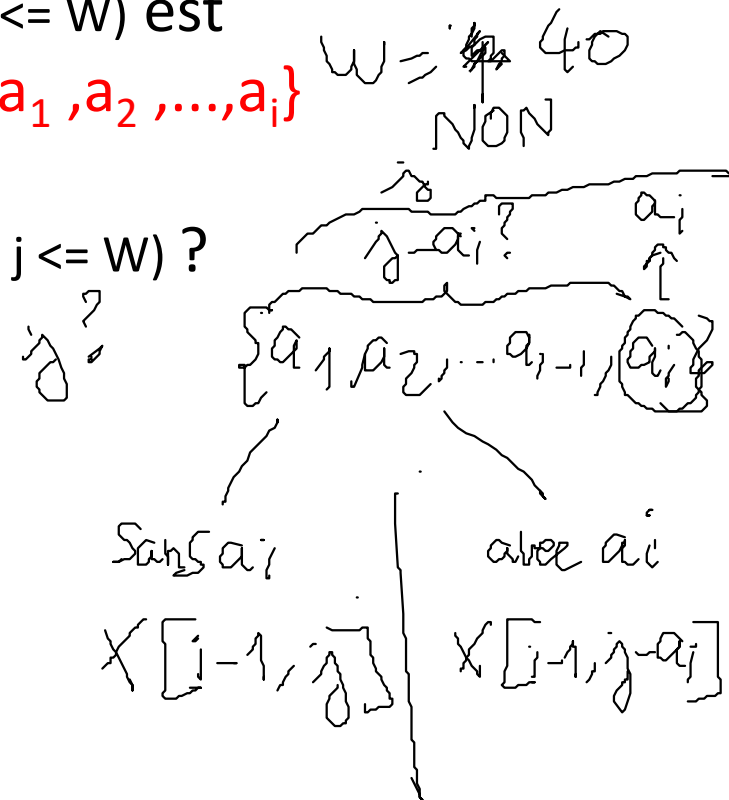
Quelle égalité est VRAI ( $2 \leq i \leq n$  et  $a_i \leq j \leq W$ ) ?

A.  $X[i, j] = X[i - 1, j] \vee X[i, j - a_i]$

**B.  $X[i, j] = X[i - 1, j] \vee X[i - 1, j - a_i]$**

C.  $X[i, j] = X[i - 1, j] \wedge X[i, j - a_i]$

D.  $X[i, j] = X[i - 1, j] \wedge X[i - 1, j - a_i]$



## Quiz

Un programme dynamique pour résoudre SubsetSum utilise un tableau de booléens  $X$  à  $n$  lignes et  $W+1$  colonnes t.q.

$X[i, j]$  (avec  $1 \leq i \leq n$ ,  $0 \leq j \leq W$ ) est Vrai ssi il existe un sous-ensemble de  $\{a_1, a_2, \dots, a_i\}$  dont la somme des éléments est  $j$ .

Quelle case du tableau X, si elle est = VRAI

indique qu'il existe un sous-ensemble de  $\{a_1, \dots, a_m\}$  dont la somme est  $W$  ?

- A.  $X[1, W]$
- B.  $X[n, 0]$
- ☒ C.  $X[n, W]$
- D.  $X[n-1, n]$

$$K = \begin{pmatrix} & \\ & W \\ & \\ & \square \end{pmatrix}$$



# Résumé

- 1) Déterminer une sous-structure optimale dont on a besoin pour résoudre le problème

$Fib(n) \rightarrow$  n<sup>o</sup> de Fib  
 $OPT(i)$  - valeur d'une sol. opt. avec les intervalles de 1 à  $i$ .  
 $X[i, j]$  -  $\exists$ ? s-ensemble de  $\{a_1, \dots, a_i\}$  de somme  $j$ ?

- 2) Caractériser (par une équation) cette sous-structure optimale

$Fib(i) = Fib(i-1) + Fib(i-2)$   
 $OPT(i) = \max \{ v_i + OPT[der(i)], OPT(i-1) \}$   
 $X[i, j] = X[i-1, j] \text{ ou } X[i-1, j - a_i]$

- 3) En déduire la valeur d'une solution optimale

$Fib(n)$ ;  $OPT(n)$ ;  $X[n, W]$

- 4) Ecrire un algo. de programmation dynamique

- récursif (avec mémorisation), ou
- itératif (approche du bas vers le haut)

- 5) Pour retrouver la solution optimale à partir de sa valeur : retour en arrière (backtrack)

# Résumé

- 1) Déterminer une sous-structure optimale dont on a besoin pour résoudre le problème  
 $\text{Fib}(n)$  :  $n^{\text{ème}}$  nombre de Fibonacci  
 $\text{OPT}(i)$  : valeur d'une solution optimale en se restreignant aux intervalles  $1, \dots, i$ .  
 $X[i, j]$  : il existe un sous-ensemble de  $\{a_1, a_2, \dots, a_i\}$  dont la somme des éléments est  $j$ .
- 2) Caractériser (par une équation) cette sous-structure optimale  
 $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$   
 $\text{OPT}(i) = \max \{ v_i + \text{OPT}(\text{der}(i)) , \text{OPT}(i-1) \}$   
 $X[i, j] = X[i-1, j] \text{ ou } X[i-1, j-a_i]$
- 3) En déduire la valeur d'une solution optimale  
 $\text{Fib}(n)$  ;  $\text{OPT}(n)$  ;  $X[n, W]$
- 4) Ecrire un algorithme de programmation dynamique pour résoudre le problème
  - récursif (avec mémorisation), ou
  - itératif (approche du bas vers le haut)
- 5) Pour retrouver la solution optimale à partir de sa valeur : retour en arrière (backtrack)

# Plus courts chemins

**Problème :** *Entrée :* un graphe  $G=(S,A)$  orienté et valué ( $c$ ) ;  
sommet origine  $s$  ; sommet destination  $v$ .  
*Sortie :* un plus court chemin de  $s$  à  $v$  dans  $G$ .  
(ou  $A(s)$ , arborescence des plus courts chemins d'origine  $s$ ).

On note  $d(x)$  la distance de  $s$  à  $x$ , pour tout  $x \in S$ .

**Propriété :** Il existe un plus court chemin entre  $s$  et  $x$  si et seulement si il n'existe pas de circuit absorbant dans un chemin entre  $s$  et  $x$  dans  $G$ .

Supposons qu'il n'y ait pas de circuit absorbant accessible à partir de  $s$ .  
Comment calculer  $A(s)$  ?

- L'algorithme de Dijkstra ne retourne pas toujours une arborescence des plus courts chemins si les coûts des arcs sont négatifs.
- Augmenter le coût des arcs de façon à avoir seulement des coûts positifs ne marche pas non plus.

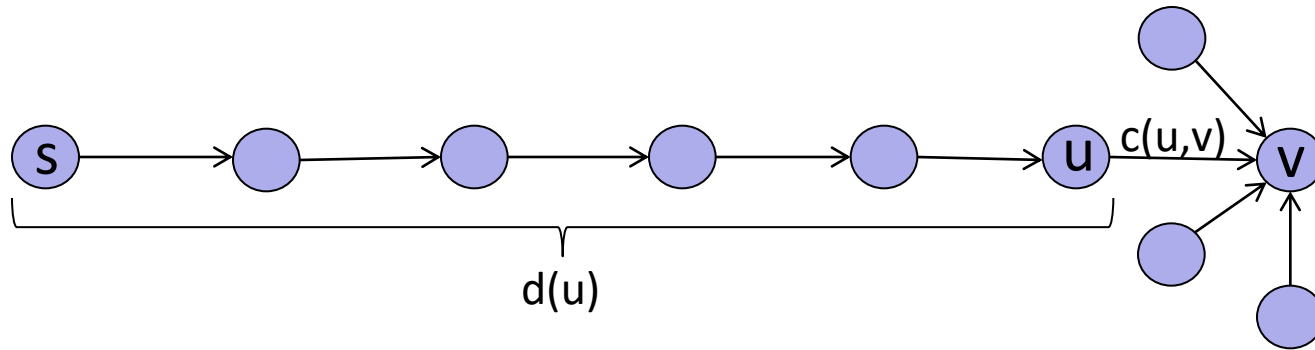


# Plus courts chemins : programme dynamique

Comment calculer le plus court chemin de  $s$  à  $v$  ?

On ne connaît pas la réponse ? On **essaie** toutes les solutions (et on prend la meilleure).

Programmation dynamique : récursion + mémorisation + essais



Quel est le dernier arc du plus court chemin entre  $s$  et  $v$  ?  
On ne sait pas  $\Rightarrow$  on les essaie tous.

$$\begin{cases} d(v) = \min_{u \mid (u,v) \in A} \{ d(u) + c(u,v) \} & \text{si } v \neq s \\ d(s) = 0 \end{cases}$$

# Plus courts chemins : programme dynamique

$$\text{On a : } \begin{cases} d(s) = 0 \\ d(v) = \min_{u \mid (u,v) \in A} \{ d(u) + c(u,v) \} \quad \text{si } v \neq s \end{cases}$$

Algorithme récursif :

```
d(s,v) :  
  si v=s retourner 0  
  d_min = + ∞  
  pour tout prédécesseur u de v faire  
    d_courant = d(s,u) + c(u,v)  
    si (d_courant < d_min)  
      d_min = d_courant  
  retourner d_min
```

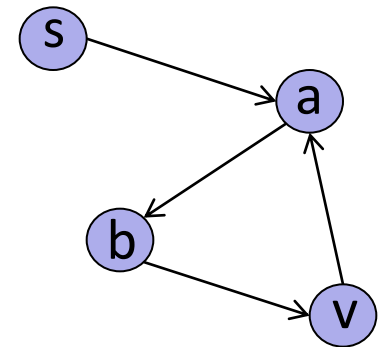
# Plus courts chemins : programme dynamique

Algorithme récursif avec mémorisation :

```
pour tout sommet i de  $S \setminus \{s\}$   
mémo[i] = vide  
mémo[s] = 0
```

```
d(s,v) :  
  si mémo[v] est vide  
    d_min =  $+\infty$   
    pour tout prédécesseur u de v faire  
      d_courant =  $d(s,u) + c(u,v)$   
      si ( $d\_courant < d\_min$ )  
        d_min = d_courant  
    mémo[v] = d_min  
  retourner mémo[v]
```

$$d(s) = 0$$
$$d(v) = \min_{u \mid (u,v) \in A} \{ d(u) + c(u,v) \}$$



Présence d'un circuit :  
l'algorithme ne termine pas.

# Plus courts chemins dans un graphe sans circuit

Cet algorithme est valide s'il n'y a **pas de circuit dans le graphe**.

```
d(s,v)
  si mémo[v] est vide
    d_min = + ∞
    pour tout prédécesseur u de v faire
      d_courant = d(s,u) + c(u,v)
      si (d_courant < d_min)
        d_min = d_courant
    mémo[v]=d_min
  retourner mémo[v]
```



# Quelle est la complexité de cet algorithme ?

```
d(s,v)
  si mémo[v] est vide
    d_min = + ∞
    pour tout prédécesseur u de v faire
      d_min = min{d_min, d(s,u) + c(u,v) }
  mémo[v]=d_min
  retourner mémo[v]
```

- A.  $O(n)$
- B.  $O(n+m)$
- C.  $O(n^2)$
- D.  $O(nm)$

Complexité de l'appel non mémorisé de  $d(v)$  :  $d^-(v) + 1$

Appels non mémorisés : un par sommet ( $n$  sommets)

=> Complexité totale =  $O(n+m)$

# Plus courts chemins dans un graphe sans circuit

Version ``du bas vers le haut`` de cet algorithme  
(on suppose que  $s$  est une racine) :

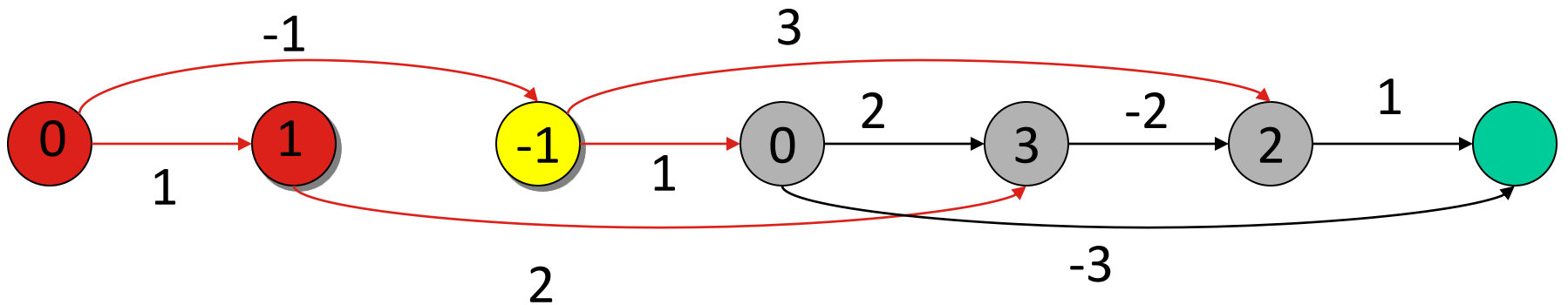
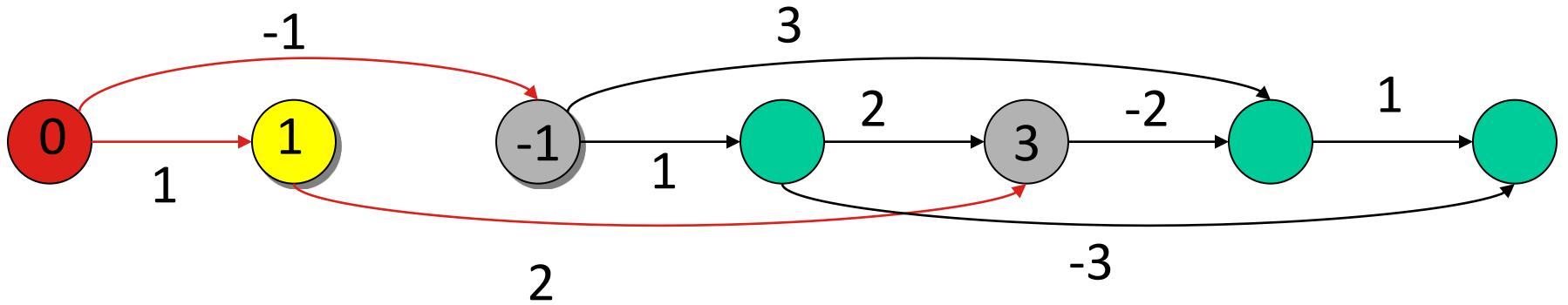
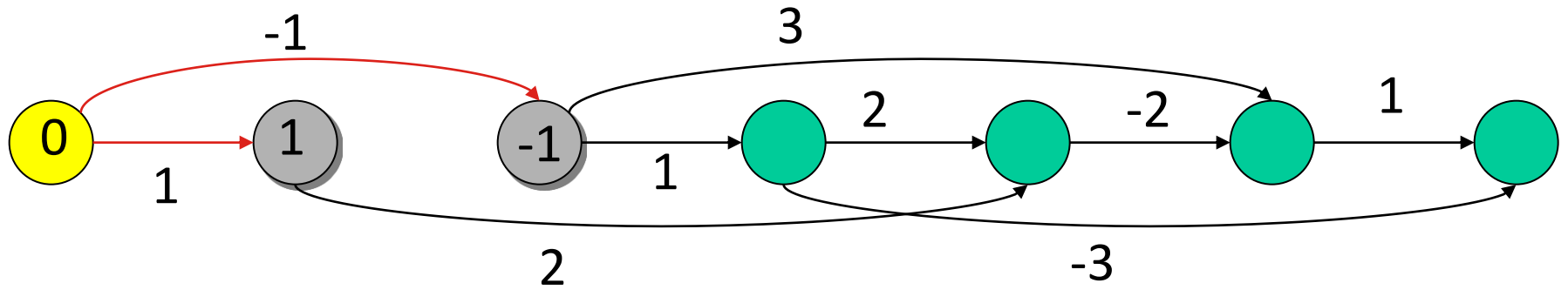
```
soit  $L=(s_1, \dots, s_n)$  une liste topologique des sommets de  $G$  telle que  $s = s_1$   
pour tout sommet  $x \neq s$   $d[x] = +\infty$  ;  $d[s] = 0$   
pour  $k$  allant de 1 à  $n$   
    pour tout successeur  $y$  de  $s_k$  faire  
        si  $d(y) > d(s_k) + c(s_k, y)$  alors  
             $d(y) = d(s_k) + c(s_k, y)$ 
```

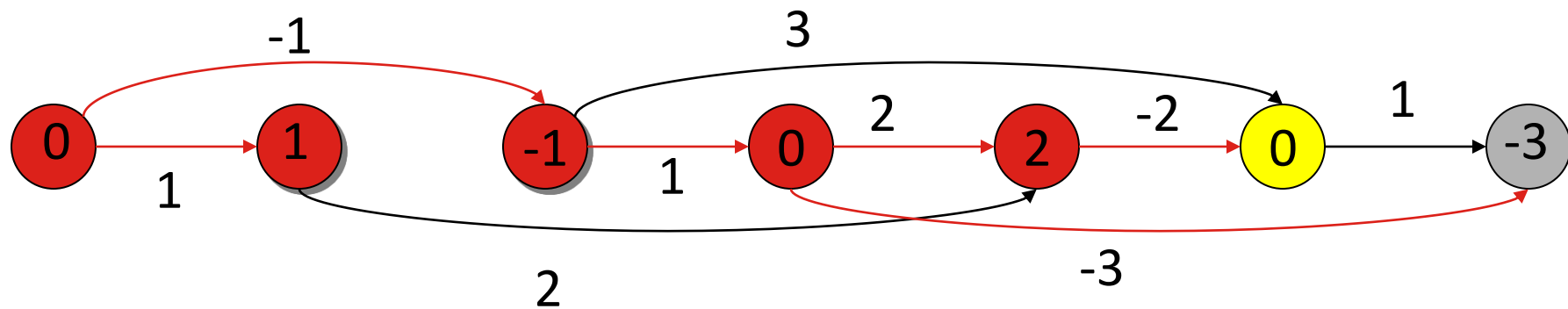
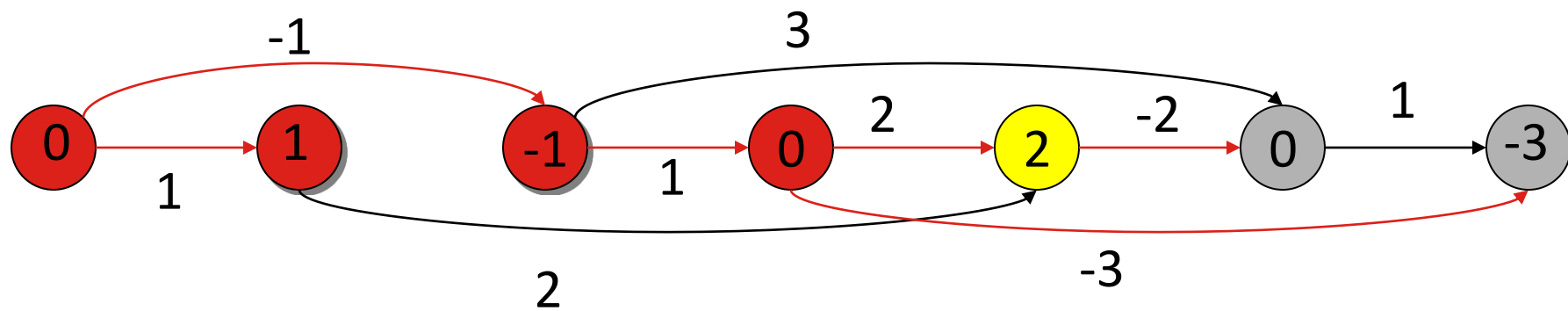
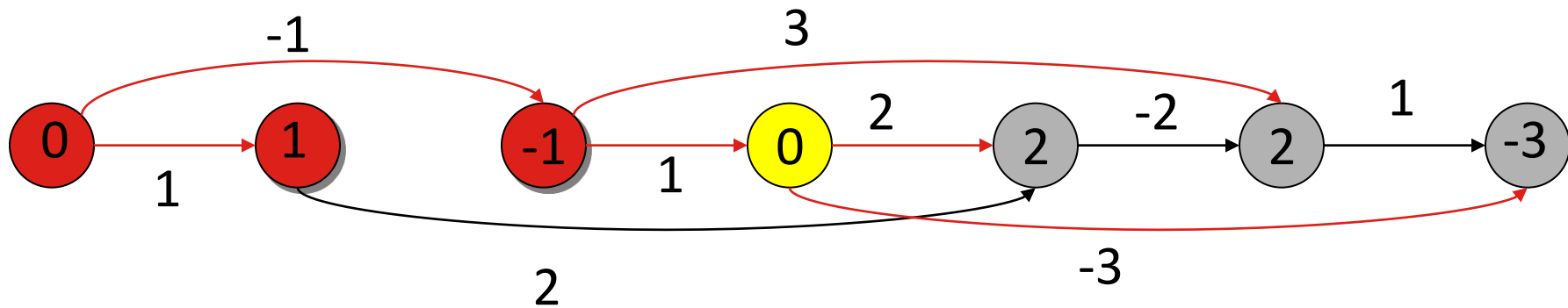
On obtient  $A(s)$ , une arborescence des plus courts chemins d'origine  $s$ .

C'est **l'algorithme de Bellman**.

Complexité totale =  $O(n+m)$

Exemple :





# Plus courts chemins dans le cas général

Les dépendances entre sous-problèmes doivent être acycliques.

Partant d'un graphe avec circuit, il faut trouver un moyen de le rendre acyclique...

Solution : on duplique  $n$  fois le graphe ( $n$  niveaux). A chaque fois que l'on suit un arc, on va vers un sommet du graphe du niveau suivant.

Exemple : (au tableau)

Cette transformation rend tout graphe acyclique.

# Plus courts chemins dans le cas général

Soit  $d_k(v)$  le coût d'un plus court chemin de  $s$  à  $v$  qui utilise au plus  $k$  arcs ( $0 \leq k \leq n-1$ ).

$$\begin{cases} d_k(v) = \min_{u \mid (u,v) \in A} \{ d_{k-1}(u) + c(u,v) \} & \text{si } v \neq s \\ d_k(s) = 0 \end{cases}$$

**But** : calcul de la valeur  $d_{n-1}(v)$  (plus court chemin élémentaire).

**Algorithme de Bellman-Ford** : algorithme récursif avec mémoïsation correspondant à la relation de récurrence ci-dessus.

Complexité de l'appel non mémoisé de  $d_k(v)$  :  $d^-(v) + 1$

Complexité :  $O(n^2 + nm)$

# Algorithme de Bellman-Ford

// Initialisation

**pour tout** sommet  $x$  **faire**

**si**  $x=s$  **alors**  $d(x) = 0$ ;  $pred=null$

**sinon**  $d(x) = +\infty$  ;  $pred=null$

// Boucle principale

**pour**  $k = 1$  à  $n-1$  **faire**

**pour** chaque arc  $(x,y)$  **faire**

**si**  $d(y) > d(x) + c(x,y)$  **alors**

$d(y) = d(x) + c(x,y)$

$pred(y) = x$

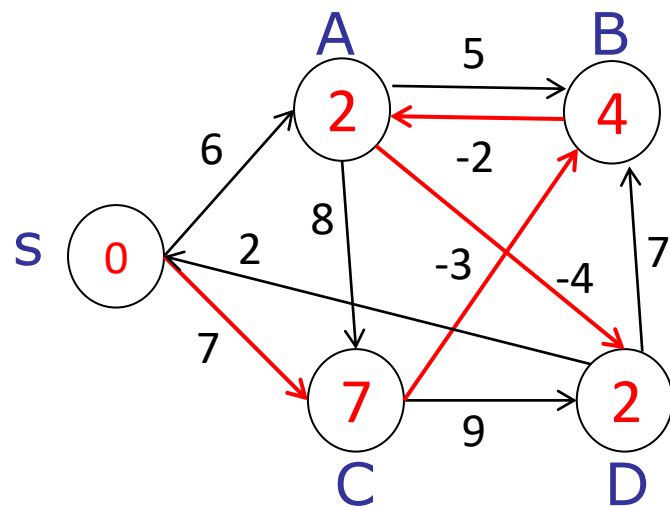
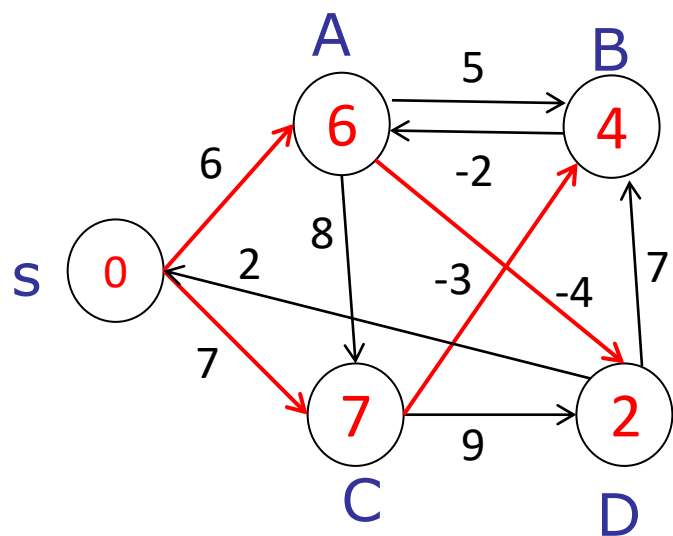
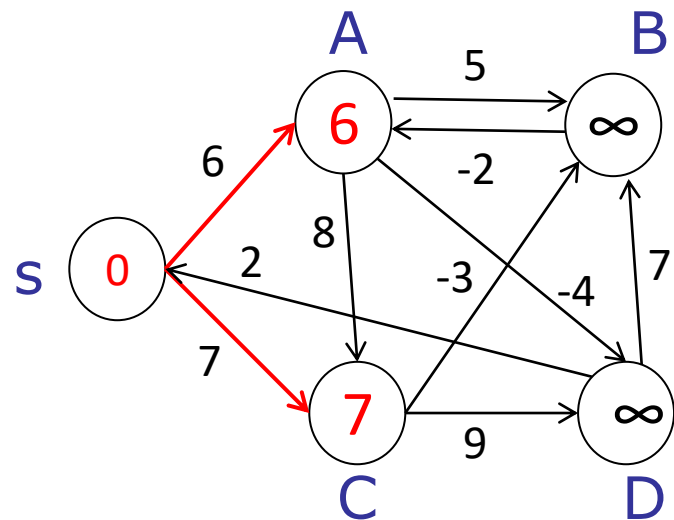
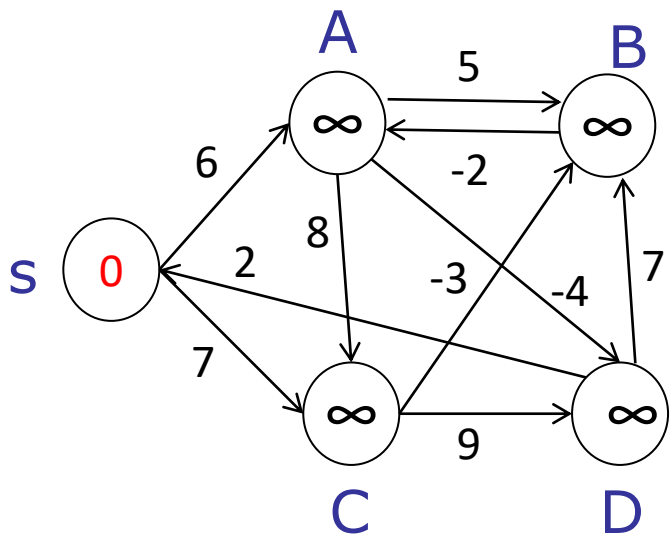
// Détection de circuit absorbant

**pour** chaque arc  $(x,y)$  **faire**

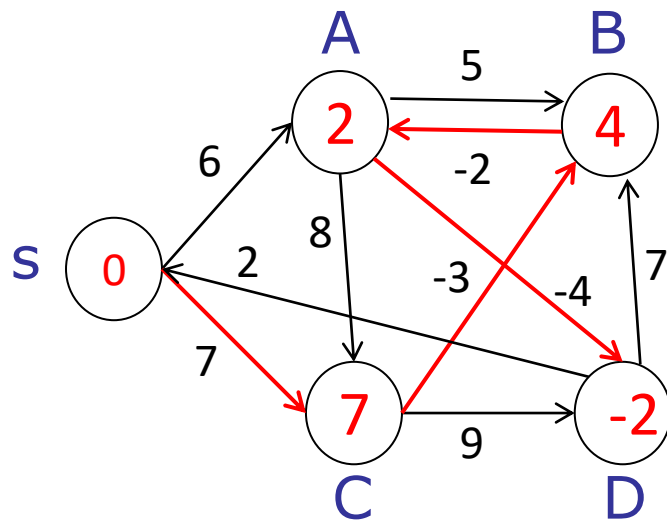
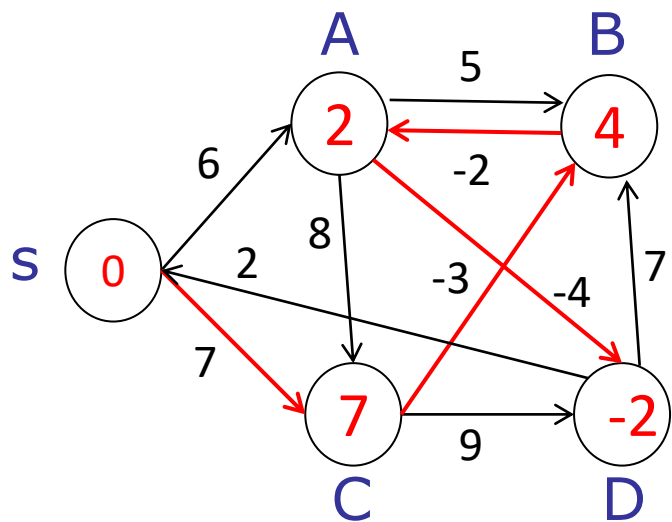
**si**  $d(y) > d(x) + c(x,y)$  **alors**

**erreur** “il n'existe pas de plus court chemin entre  $s$  et  $x$ ”

# Exemple







# Validité

**Propriété 1 :** A tout moment après l'étape d'initialisation, si  $d(x) \neq \infty$ ,  $d(x)$  est égal à la longueur d'un chemin de  $s$  à  $x$ .

*Preuve (récurrence sur le nombre  $i$  de mises à jour de valeurs  $d(\cdot)$ ) :*

- Pour  $i=0$ , la propriété est vérifiée.
- Supposons que cette propriété soit vérifiée jusqu'à la  $(i-1)$ -ème mise à jour, et que la  $i$ -ème mise à jour consiste à modifier  $d(x)$  après l'examen de l'arc  $(y,x)$ .

Par hypothèse de récurrence,  $d(y)$  est la longueur d'un chemin  $\mu$ , de  $s$  à  $y$ . Ainsi  $d(x) = d(y) + c(y,x)$  est la longueur du chemin de  $s$  à  $x$  qui est  $\mu.(y,x)$ .

# Validité

## Propriété 2 : (invariant de boucle)

*Après  $i$  itérations de la boucle “pour” principale : s’il existe un chemin de  $s$  à  $x$  d’au plus  $i$  arcs, alors  $d(x)$  est inférieur ou égal à la longueur du plus court chemin ayant au plus  $i$  arcs de  $s$  à  $x$ .*

## Preuve (récurrence sur $i$ )

- Pour  $i=0$ , l’invariant est vérifié.
- Supposons qu’il soit vérifié jusqu’au rang  $i-1$ 
  - Soit  $\mu$  un plus court chemin (pcc) d’au plus  $i$  arcs de  $s$  à  $x$ . Soit  $y$  le dernier sommet avant  $x$  sur  $\mu$  : le chemin de  $s$  à  $y$  est un pcc de  $s$  à  $y$  d’au plus  $(i-1)$  arcs. Par hyp. d’induction, après  $(i-1)$  itérations,  $d(y)$  est  $\leq$  à la longueur de ce chemin.
  - A l’itération  $i$ ,  $d(x)$  est comparé à  $d(y)+c(x,y)$  et mise à jour à cette valeur si cela diminue  $d(x)$ . Donc, à la fin de cette itération,  $d(x)$  est  $\leq$  à la longueur de  $\mu$ .

# Validité

**Propriété 1 :** A tout moment après l'étape d'initialisation, si  $d(x) \neq \infty$ ,  $d(x)$  est égal à la longueur d'un chemin de  $s$  à  $x$ .

**Propriété 2 :**

*Après  $i$  itérations de la boucle “pour” principale :* s'il existe un chemin de  $s$  à  $x$  d'au plus  $i$  arcs, alors  $d(x)$  est inférieur ou égal à la longueur d'un plus court chemin ayant au plus  $i$  arcs de  $s$  à  $x$ .

=> à la fin de la boucle principale, pour tout sommet  $x$ , s'il existe un chemin de  $s$  à  $x$  dans  $G$ , alors  $d(x)$  est égal à la longueur d'un plus court chemin de  $s$  à  $x$ .

# Complexité

```
// Initialisation
// Boucle principale :
pour k = 1 à n-1 faire
    pour chaque arc (x,y) faire
        si  $d(y) > d(x) + c(x,y)$  alors
             $d(y) = d(x) + c(x,y)$ 
             $\text{pred}(y) = x$ 
// Détection de circuit absorbant
pour chaque arc (x,y) faire
    si  $d(y) > d(x) + c(x,y)$  alors
        erreur "il n'existe pas de plus court chemin entre s et x"
```

- Complexité :  $O(nm)$
- **Remarque** : si à l'itération  $i$  aucune valeur  $d(.)$  n'est modifiée, l'algorithme peut s'arrêter.