Algorithmique
Cours 5 : Programmation dynamique

ROB3 – année 2017-2018

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 (qui ne se chevauchent pas), 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 (qui se chevauchent), et cherche (et stocke) des solutions de sous-problèmes de plus en plus grands

Bref historique

- Programmation dynamique : paradigme développé par **Richard Bellman** en 1953 chez RAND Corporation.
- « Programmation » = planification
- Technique de conception d'algorithme très générale et performante.
- Permet de résoudre de nombreux problèmes d'optimisation.

Richard Bellman

Pourquoi « programmation dynamique » ?

« *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?* »

Richard Bellman (1984)

Revisitons Fibonacci...

Soit F_n = nombre de lapins au mois n

$F_1 = 1$

$F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$

Ce sont les nombres de Fibonacci :

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

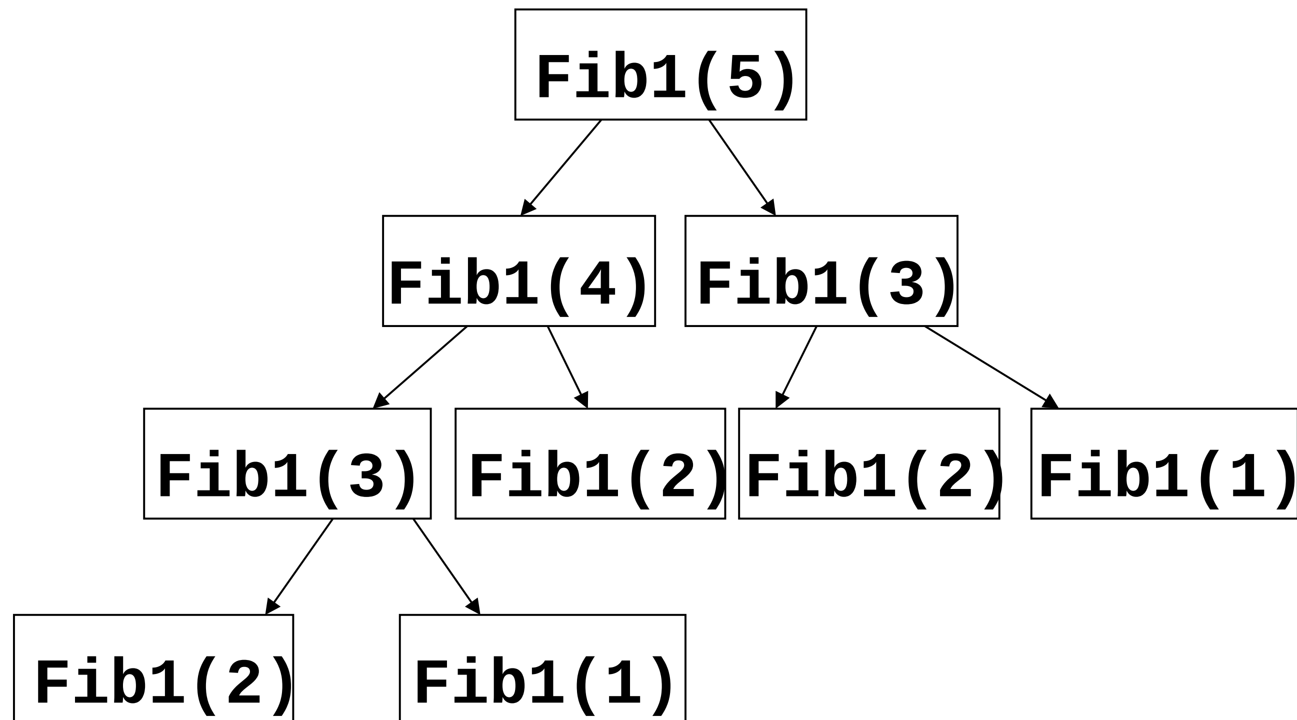
Ils croissent *très* vite: $F_{30} > 10^6$!

En fait, $F_n \approx 2^{0.694n}$, croissance exponentielle.

Leonardo da Pisa, dit Fibonacci

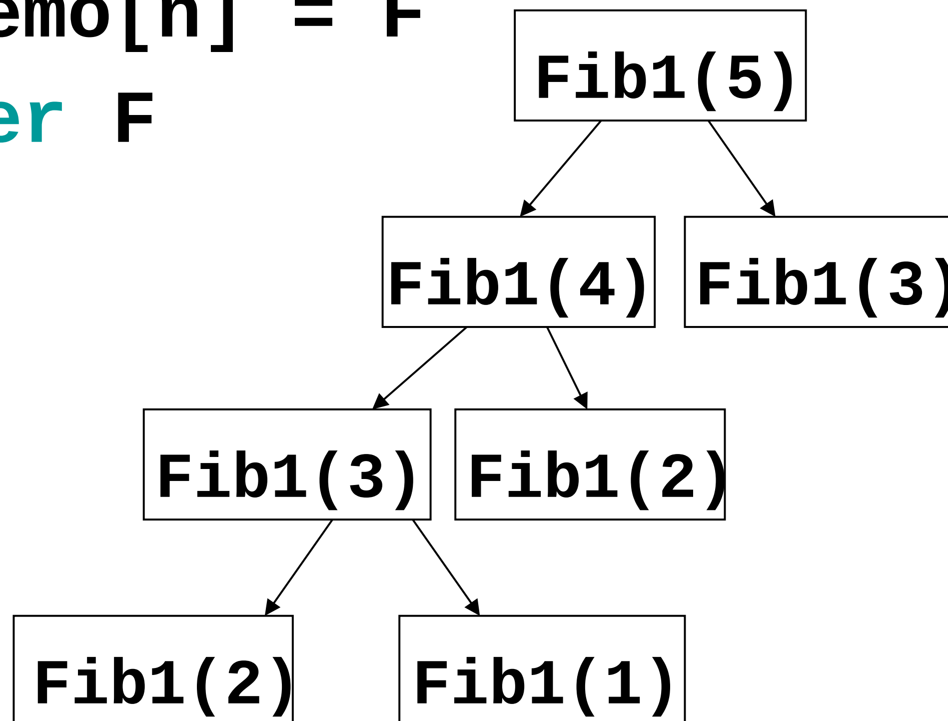
Algorithme récursif inefficace

```
fonction Fib1(n)
si n = 1 retourner 1
si n = 2 retourner 1
retourner Fib1(n-1) + Fib1(n-2)
```



Algorithme récursif avec *mémoïsation*

```
fonction Fib1mem(n)
si mémo[n] est défini retourner mémo[n]
si n ≤ 2 alors F = 1
sinon F = Fib1mem(n-1) + Fib1mem(n-2)
      mémo[n] = F
retourner F
```



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.*

Analyse de complexité

```
fonction Fib1mem(n)
si mémo[n] est défini retourner mémo[n]
si n ≤ 2 alors F = 1
sinon F = Fib1mem(n-1) + Fib1mem(n-2)
      mémo[n] = F
retourner F
```

Fib1mem(k) induit des appels récursifs seulement la première fois qu’elle est appelée.
Nombre d'appels non mémoïsés : n.
Temps d'un appel (sans compter les appels récursifs non mémoïsés) : $O(n)$ du fait de l'addition entière Fib1mem(k-1) + Fib1mem(k-2).
(mémo[i] est retourné en $\Theta(1)$ si mémo est un tableau.)
Complexité temporelle : $\Theta(n^2)$.

Plus généralement

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

Complexité temporelle :

nombre de sous-problèmes

x

complexité par sous-problème*

* : on ne compte pas les appels récursifs.

Approche « du bas vers le haut »

Dans cette approche, on remplit un tableau :

```
fonction Fib2(n)
 Créer un tableau fib[1..n]
 fib[1] = 1
 fib[2] = 1
 pour i = 3 à n:
     fib[i] = fib[i-1] + fib[i-2]
 retourner fib[n]
```

- Mêmes calculs que dans la version mémoïsée. Il faut toutefois identifier un ordre dans lequel résoudre les sous-problèmes.
- Complexité temporelle : nombre de cellules du tableau x complexité de calcul d'une cellule.
- Permet souvent de baisser la complexité spatiale.

Complexité spatiale : exemple

```
fonction Fib2opt(n)
 fib' = 1
 fib = 1
 pour i = 3 à n:
     temp = fib
     fib = fib + fib'
     fib' = temp
 retourner fib
```

- Complexité spatiale `Fib2` : $O(n^2)$ car tableau de n cases avec des entiers codés sur au plus n bits.
- Complexité spatiale `Fib2opt` : O(n) car les variables `fib` et `fib'` comportent des entiers codés sur au plus n bits.

Concevoir une procédure de programmation dynamique

Quatre étapes :

- Définir les sous-problèmes.
- Identifier une relation de récurrence entre les solutions des sous-problèmes.
- En déduire un algorithme récursif avec mémoïsation ou une approche du bas vers le haut
- Résoudre le problème original à partir des solutions des sous-problèmes

Huit américain

But du jeu

Etre le premier joueur à se défausser de toutes ses cartes.

Déroulement du jeu

Les joueurs jouent chacun à leur tour dans le sens des aiguilles d'une montre (au début du jeu en tous cas). Celui qui commence est à la gauche du donneur. A son tour, le joueur à le choix de jouer (c'est à dire de poser UNE carte sur le talon) soit une carte de la même couleur que celle qui est en haut du talon, soit une carte de la même valeur, soit un huit à tout moment (et choisir la nouvelle couleur à jouer). On dit alors que les cartes sont compatibles. Lorsqu'un joueur ne peut pas jouer de carte, il pioche une carte et passe son tour.

Par exemple, si la première carte du talon est un 4 de coeur, le joueur peut jouer soit n'importe quel coeur, soit n'importe quel 4, soit n'importe quel huit.

Huit américain

Données : une séquence de cartes c[1] … c[n]

Exemple : 7♣ 7♥ K♣ K♠ 8♥

But : trouver la plus longue sous-séquence $c[i_1]$ … $c[i_k]$ ($i_1 < i_2 < \ldots < i_k$) où $c[i_j]$ et $c[i_{j+1}]$ sont compatibles (noté $c[i_j] \sim c[i_{j+1}]$) pour j=1,…,k-1.

Exemple :

7♣ K♣ K♠ 8♥ est la plus longue sous-séquence pour l'exemple ci-dessus.

Conception de l'algorithme

- Sous-problème : soit opt(i) la longueur de la plus longue sous-séquence débutant par c[i]
- Question : comment relier la valeur de opt(i) avec opt(i+1),…,opt(n) ?
- Relation de récurrence :

  $\text{opt}(i) = 1 + \max \{0, \text{opt}(j) \text{ pour } j>i, c[i] \sim c[j]\}$

- Problème original :

  $\max\{\text{opt}(i) \text{ pour } i=1,\ldots,n\}$

Algorithme récursif avec mémoïsation

Initialisation

pour i allant de 1 à n

mémo[i] = vide

mémo[n] = 1

Calcule_OPT(i)

si mémo[i] est vide

mémo[i] = 1+max {0,Calcule_OPT(j) pour j>i, c[i]$\sim$c[j]}

retourner mémo[i]

Problème original

max {Calcule_OPT(i) pour i=1,...n}

Complexité : Initialisation : $\Theta(n)$

Calcule_OPT(i) : n x O(n) $\Rightarrow$ $O(n^2)$

$O(n^2)$ Problème original à partir des sous-pbs : $\Theta(n)$

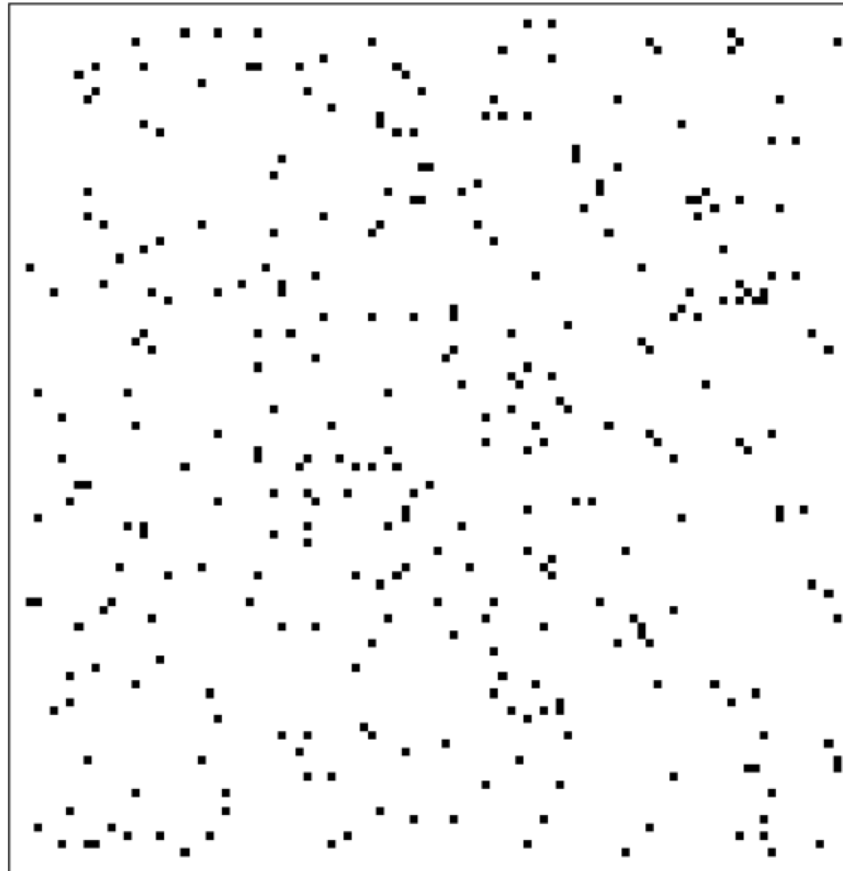
Approche du bas vers le haut

pour i allant de n à 1
    mémo[i] = 1+max {0, mémo[j] pour j>i, c[i]∼c[j]}
retourner max {mémo[i] pour i=1,...,n}

Complexité : n problèmes x O(n)
    opération de max en dernière ligne : O(n)
    ⇒ $O(n^2)$

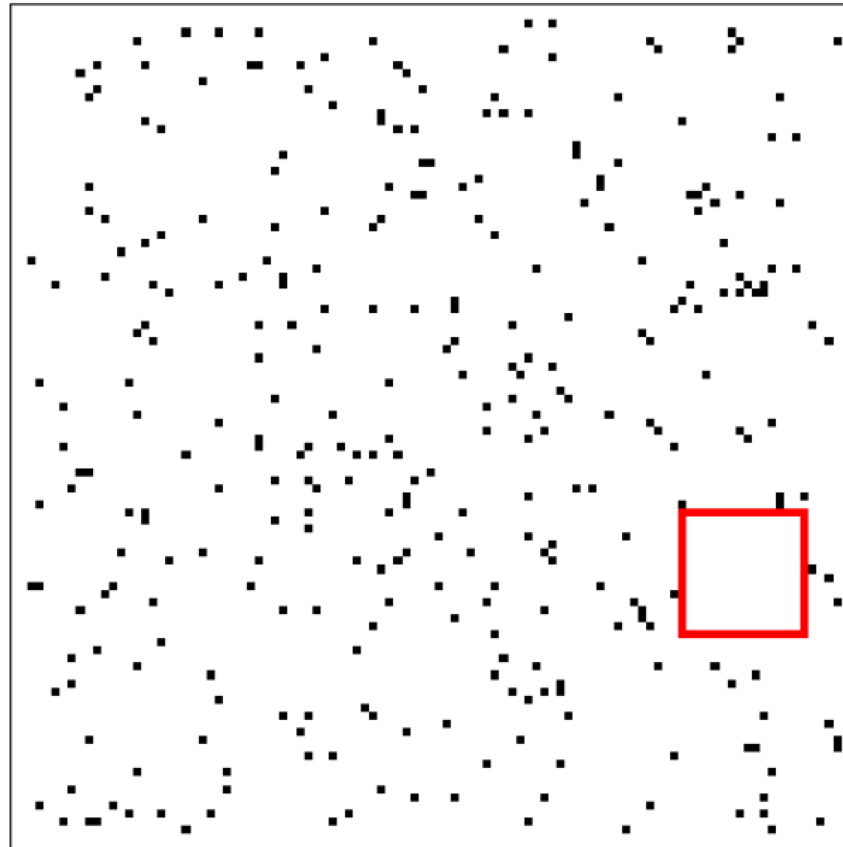
Plus grand carré blanc

On considère le problème suivant : étant donné une image monochrome *n x n*, déterminer le plus grand carré blanc, i.e. qui ne contient aucun point noir.



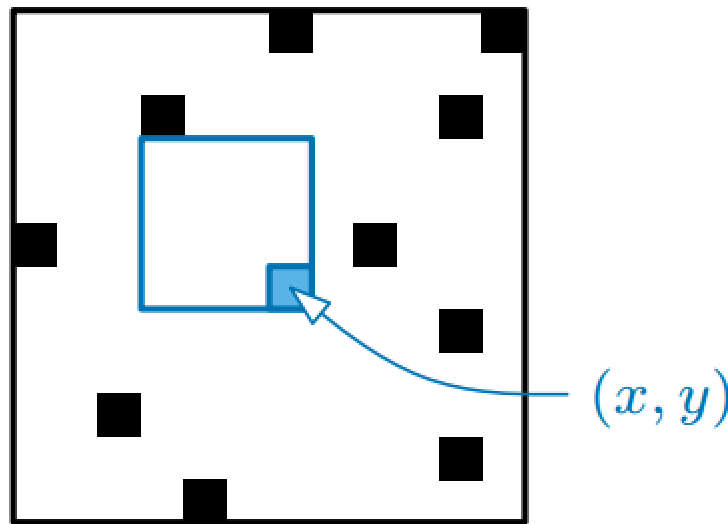
Plus grand carré blanc

On considère le problème suivant : étant donné une image monochrome *n x n*, déterminer le plus grand carré blanc, i.e. qui ne contient aucun point noir.



Sous-problème

Déterminer la taille PGCB(x,y) du plus grand carré blanc dont le pixel en bas à droite a pour coordonnées (x,y)

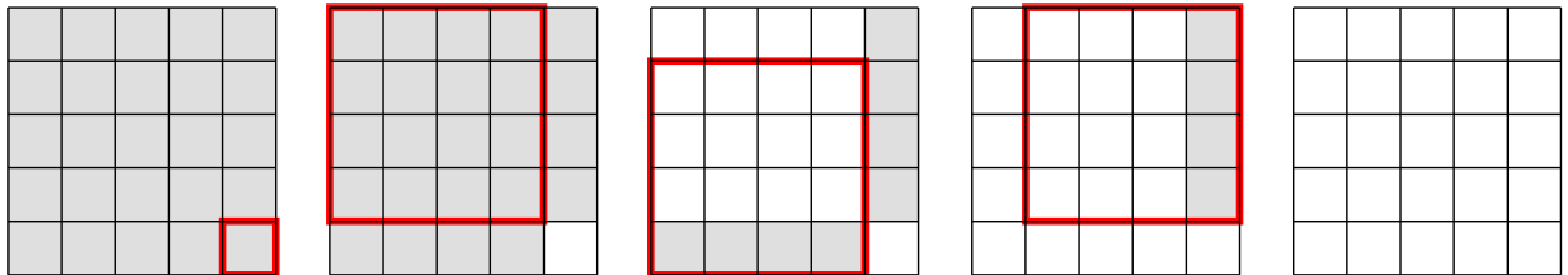


Observation clé

Un carré m x m de pixels C est blanc si et seulement si :

- le pixel en bas à droite de C est blanc ;
- Les trois carrés (m-1) x (m-1) en haut à gauche, en haut à droite et en bas à gauche sont tous blancs.

Preuve par l'image :



Relation de récurrence

- Si le pixel (x,y) est noir, alors :

PGCB(x,y)=0.

- Si (x,y) est blanc et dans la première ligne en haut ou la première colonne à gauche, alors :

PGCB(x,y)=1.

- Si (x,y) est blanc et ni dans la première ligne en haut ni dans la première colonne à gauche, alors :

PGCB(x,y) = 1 + min{PGCB(x-1,y-1),PGCB(x,y-1),PGCB(x-1,y)}.

Algorithme récursif avec mémoïsation

```
fonction PGCB(x,y)
si mémo[x,y] est défini retourner mémo[x,y]
si (x,y) est noir retourner 0
si x = 1 ou y = 1 retourner 1
sinon
   mémo[x,y] = 1 + min{PGCB(x-1,y-1),PGCB(x,y-1),PGCB(x-1,y)}
  retourner mémo[x,y]
```

Analyse de complexité.

Nombre d'appels non mémoïsés : n^2.
Temps d'un appel (sans compter les appels récursifs non mémoïsés) : Θ(1).

(mémo[x,y] est retourné en Θ(1) si mémo est un tableau.)
Complexité temporelle : **$\Theta(n^2)$**.

Version « du bas vers le haut »

```
fonction PGCB(n)
pour x = 1 à n
    pour y = 1 à n
        si (x,y) est noir
           pgcb[x,y] = 0
        sinon si x = 1 ou y = 1
           pgcb[x,y] = 1
        sinon
           pgcb[x,y] = 1 + min{pgcb(x-1,y-1),pgcb(x,y-1),pgcb(x-1,y)}
```

Analyse de complexité.

Nombre de cases du tableau : n^2.
Complexité de calcul d'une case : $\Theta(1)$.

Complexité temporelle : $\boldsymbol{\Theta(n^2)}$.

Résolution du problème de départ

La taille du plus grand carré blanc est :

$$\max_{x=1,\ldots,n}\max_{y=1,\ldots,n} \text{mémo}[x,y] \text{ (version récursive)}$$

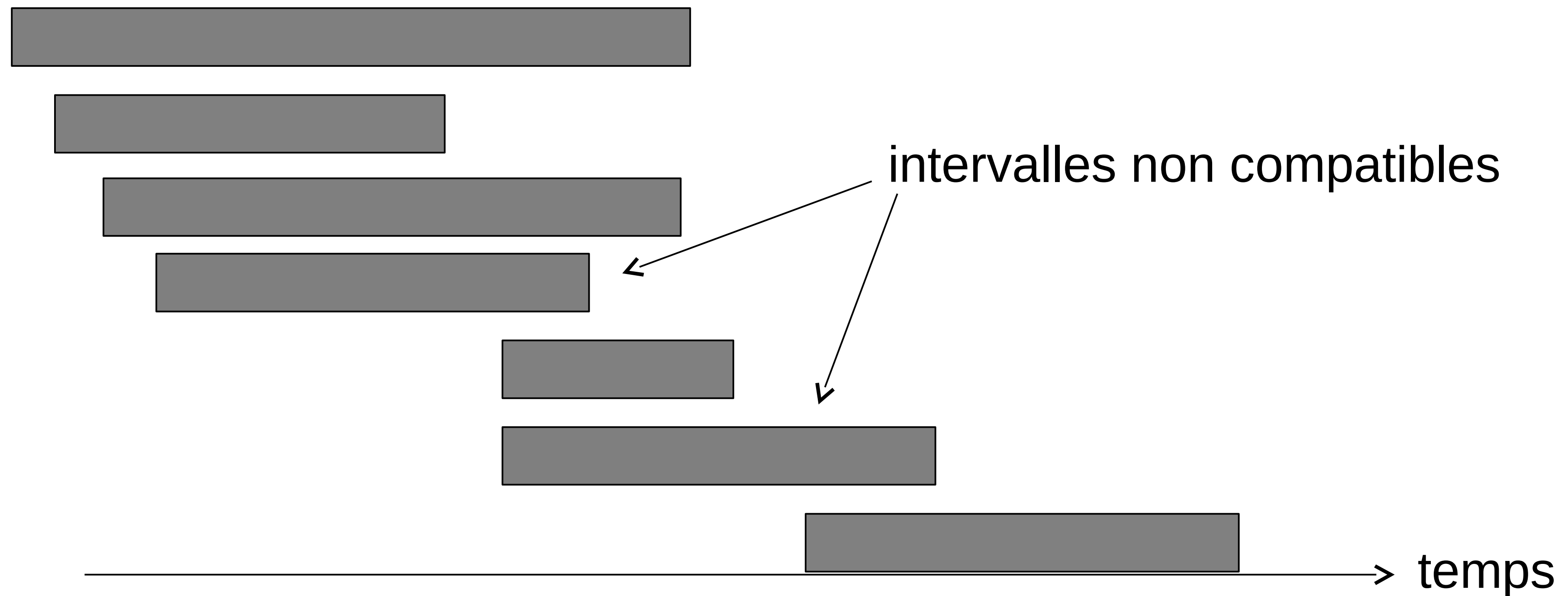
$$\max_{x=1,\ldots,n}\max_{y=1,\ldots,n} \text{pgcb}[x,y] \text{ (version itérative)}$$

Ce calcul requiert de parcourir les n^2 cases du tableau mémo ou pgcb, et se fait donc en **$\Theta(n^2)$**.

La complexité globale de l'algorithme de programmation dynamique est donc **$\Theta(n^2)$.**

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 maximum.



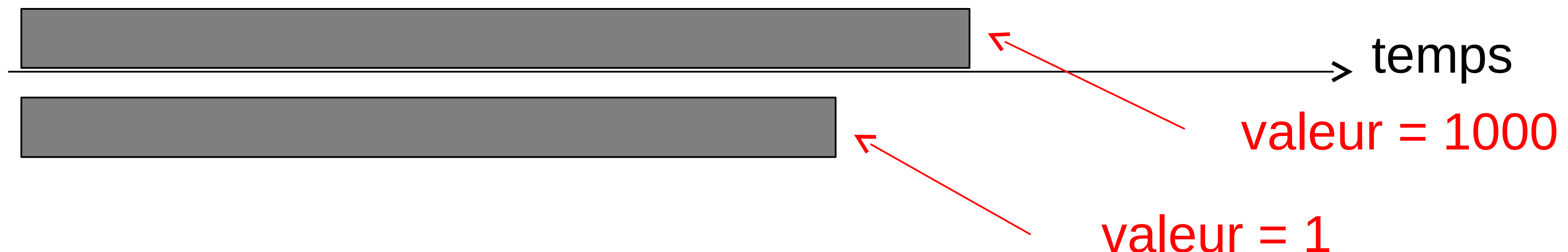
Ordonnancement d’intervalles pondérés

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 est optimal si tous les poids sont égaux à 1.

Il peut être mauvais dans la version pondérée :

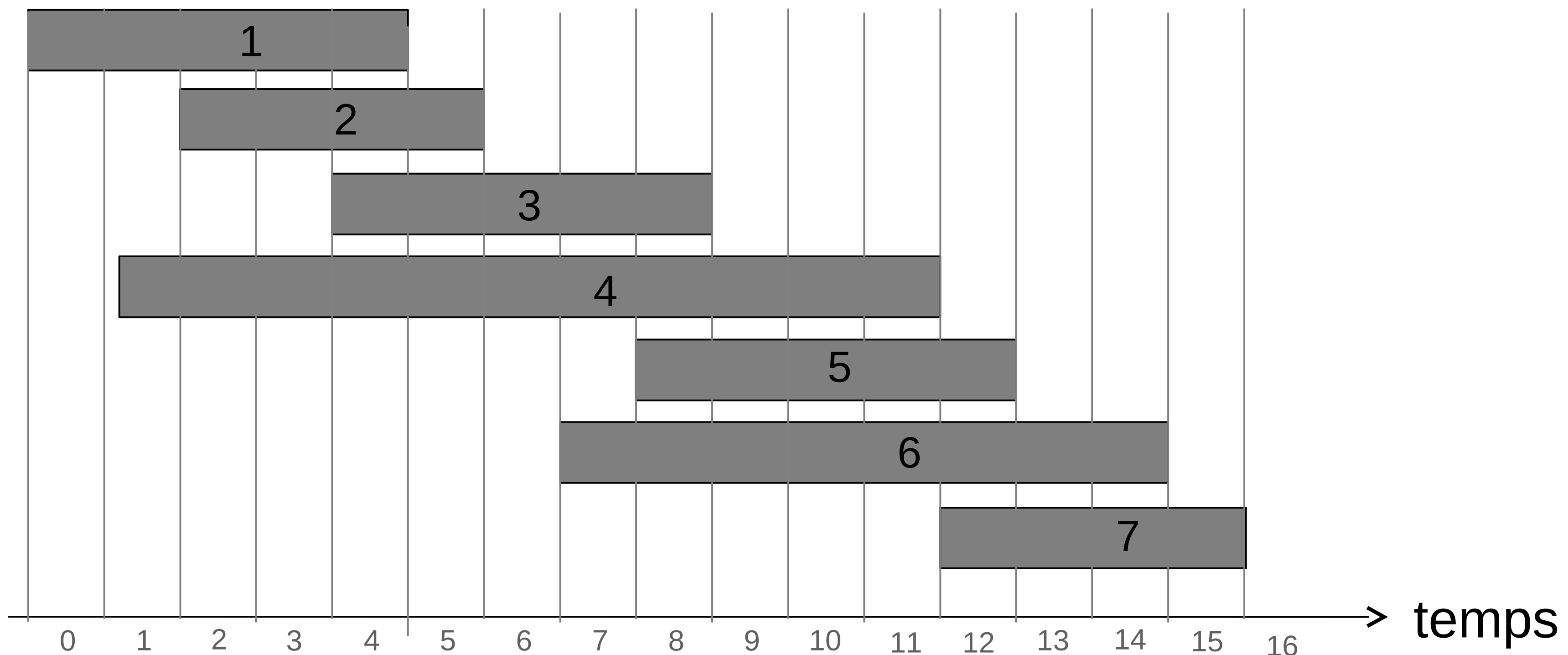


Ordonnancement d’intervalles pondérés

Notation : on numérote les intervalles par dates de fin croissante : $f_1 \leq f_2 \leq \ldots \leq f_n$

Définition : der(j) = plus grand numéro i < j tel que l’intervalle i est compatible avec l’intervalle j .

Exemple : der(7)=4 ; der(6)= 2 ; der(2)=0.



Relation de récurrence

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, …, i-1}
- On doit choisir la solution optimale du problème restreint aux intervalles 1,2,…,der(i)

Cas 2 : i n'est pas dans OPT

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

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

Algorithme récursif naïf

Entrée : n, d[1..n], f[1..n], v[1..n]
Trier les intervalles de façon à ce que f[1] ≤ f[2] ≤ … ≤ f[n].
Calculer der[1],der[2],…, der[n]

```
Calcule_OPT(i)
    si i=0  s = 0
    sinon s = max {v[i] + Calcule_OPT(der[i]) , Calcule_OPT(i-1) }
    retourner s
```

Complexité : exponentielle (cf. cas où pour tout i, der(i)=i-2)

Algorithme récursif avec mémoïsation

Entrée : n, d[1..n], f[1..n], v[1..n]
Trier les intervalles de façon à ce que f[1] ≤ f[2] ≤ … ≤ f[n].
Calculer der[1],der[2],…, der[n]

```
pour j allant de 1 à n
    mémo[j] = vide
mémo[0] = 0

Calcule_OPT(i)
  si mémo[i] est vide
    mémo[i] = max {v[i] + Calcule_OPT(der[i]),Calcule_OPT(i-1)}
  retourner mémo[i]
```

Complexité : Initialisation : Θ(n log n)
Calcule_OPT(n) : Θ(n)

Version « 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] ≤ f[2] ≤ … ≤ f[n].
Calculer der[1],der[2],…, der[n]

```
mémo[0] = 0
pour i allant de 1 à n
    mémo[i] = max { v[i] + mémo[der[i]] , mémo[i-1] }
```

Complexité : tri des intervalles : $\Theta(n \log n)$
boucle : $\Theta(n)$

Comment retrouver la solution optimale ?

```
Trouver_solution(i)
      si i = 0
                retourner ∅
      si v[i] + mémo(der[i])  > mémo[i-1]
                retourner {i} U Trouver_solution(der[i])
      sinon
                retourner Trouver_solution(i-1)
```

Complexité : Θ(n)   (nombre d'appels récursifs ≤ n)

Distance d'édition

Algorihtme

Algorithme
Algorithmie

Ignorer
Tout ignorer
Ajouter au dictionnaire
Toujours corriger en
Orthographe et grammaire

Définir la langue de la sélection
Définir la langue du paragraphe

Distance d'édition

- Comment calculer la distance d'édition entre deux chaînes de caractères s et t ?
- Distance d'édition DE(s,t) : nombre minimum d'insertions, suppressions et remplacements de caractères pour changer s en t.
- Exemple : de ANES à GENIE

  ANES – GANES – GENES – GENIES – GENIE

- Donc DE(ANES,GENIE) $\leq$ 4

  (en fait DE(ANES,GENIE) = 4)

Alignement optimal

- Problème équivalent au calcul de la distance d'édition
- Un alignement de s et t : écrire s et t l'une au dessus de l'autre, en insérant des '–' entre les lettres
- Valeur d'un alignement : nombre de positions où les deux chaînes obtenues diffèrent
- Exemple :

  – A N – E S
  G E N I E –

- Remarque : dans un alignement optimal, il n'est jamais nécessaire d'avoir deux '–' dans une même colonne.

Relation de récurrence

- Soit n la longueur de s, et m la longueur de t
- Soit s[k] le k[ème] caractère de s, et s[1...k] la sous-chaîne formé des k premiers caractères de s
- Dans un alignement optimal de s et t non vides, trois cas pour le dernier couple de caractères :

s[n]	s[n]	–
t[m]	–	t[m]

DE(s,t) = min {DE(s[1...n-1],t[1...m-1])+[s[n] ≠ t[m]],
DE(s[1...n-1],t)+1,
DE(s,t[1...m-1])+1}

Algorithme récursif avec mémoïsation

Initialisation

```
    pour i allant de 0 à n
        pour j allant 0 à m
            mémo[i,j] = vide
            si i = 0 alors mémo[0,j] = j
            si j = 0 alors mémo[i,0] = i
```

Calcule_DE(s,t,i,j)

```
    si mémo[i,j] est vide
        mémo[i,j] = min {Calcule_DE(s,t,i-1,j-1)+[s[i] ≠ s[j]],
                         Calcule_DE(s,t,i-1,j)+1,
                         Calcule_DE(s,t,i,j-1)+1}
    retourner mémo[i,j]
```

Problème original

```
    Calcule_DE(s,t,n,m)
```

Complexité : Initialisation : $\Theta(nm)$

Calcule_DE(s,t,n,m) : nm x $\Theta(1) \Rightarrow \Theta(nm)$

$\Theta(nm)$ Problème original à partir des sous-pbs : $\Theta(1)$

Version « du bas vers le haut »

Entrée : chaînes de caractères s et t, de longueurs n et m

```
pour j allant de 0 à m
    pour i allant de 0 à n
            si i = 0 alors mémo[i,j] = j
            sinon si j = 0 alors mémo[i,j] = i
            sinon
                    mémo[i,j] = min{mémo[i-1,j-1] + [s[i]≠s[j]],
                                    mémo[i-1,j]+1,
                                    mémo[i,j-1]+1}
retourner mémo[n,m]
```

Complexité : nm sous-problèmes x $\Theta(1) \Rightarrow \Theta(nm)$

Exemple

		G	E	N	I	E
	0	1	2	3	4	5
A	1	1	2	3	4	5
N	2	2	2	2	3	4
E	3	3	2	3	3	3
S	4	4	3	3	4	4