

III – Programmation dynamique

M. Solnon

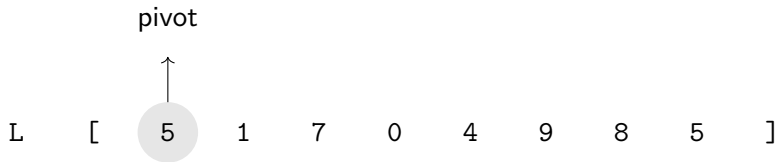
PT, Lycée Gustave Eiffel, Dijon

6 octobre 2026

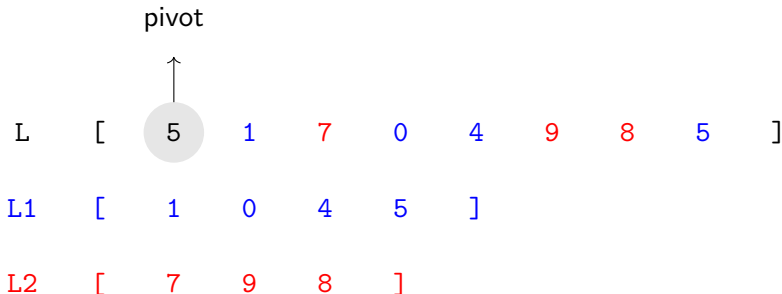
Diviser pour régner : tri rapide

```
L [ 5 1 7 0 4 9 8 5 ]
```

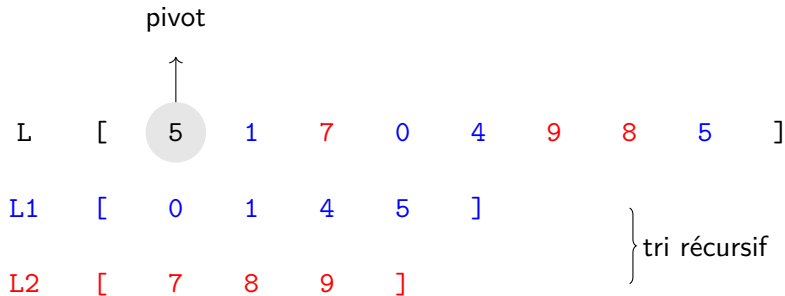
Diviser pour régner : tri rapide



Diviser pour régner : tri rapide



Diviser pour régner : tri rapide



Diviser pour régner : tri rapide



Diviser pour régner : tri rapide

Fonction TRI_RAPIDE(L) :

Si L a au plus 1 élément **Alors**

Renvoyer L

Sinon

 pivot $\leftarrow$ L[0]

 L1 $\leftarrow$ [x $\in$ L[1:] tq. x $\leq$ pivot]

 L2 $\leftarrow$ [x $\in$ L[1:] tq. x $>$ pivot]

 L1 $\leftarrow$ TRI_RAPIDE(L1)

 L2 $\leftarrow$ TRI_RAPIDE(L2)

Renvoyer Concaténation de L1, [pivot], L2

Fin Fonction

Rendu de monnaie

Exemple réel

Comment rendre 39 € ?

Exemple plus intéressant

Et avec uniquement des unités de valeur 1 €, 3 € et 4 € ?

Plusieurs solutions possibles

$$39 \text{ €} = 39 \times 1 \text{ €} = 9 \times 4 \text{ €} + 3 \text{ €}$$

Problème du rendu de monnaie

Trouver une décomposition compatible en le moins de pièces possible.

Notation

$$\mathcal{M}(k) = \min \left\{ \sum_{i=0}^{n-1} a_i \mid (a_0, \dots, a_{n-1}) \in \mathbb{N}^n \text{ tq. } \sum_{i=0}^{n-1} a_i p_i = k \right\}$$

Algorithme glouton (1)

Idée générale

Prendre localement une solution qui semble « optimale »

Division euclidienne de k par p_i (en python)

Quotient : `k // p[i]`

Reste : `k % p[i]`

Principe ici

Prendre le maximum de pièces de la plus grande valeur

Algorithme glouton (2)

Fonction monnaieGlouton(k, pieces):

 aRendre $\leftarrow$ k

 nbPieces $\leftarrow$ 0

Pour chaque piece **De** pieces[::-1], **Faire**

 nbPieces $\leftarrow$ nbPieces + (quotient de aRendre
 $\div$ piece)

 aRendre $\leftarrow$ reste de aRendre $\div$ piece

Renvoyer nbPieces

Fin Fonction

Exercice

Consigne

Traduire ceci en code Python.

Test

Vérifier que `monnaieGlouton(6, [1,3,4])` renvoie 3.

Analyse du problème

Idée fondamentale

Considérer la dernière pièce rendue

Exemple : $k = 6$, $pieces = [1, 3, 4]$

- Si on rend 1, il reste 5 à rendre
- Si on rend 3, il reste 3 à rendre
- Si on rend 4, il reste 2 à rendre

$$\mathcal{M}(6) = 1 + \min \{ \mathcal{M}(5), \mathcal{M}(3), \mathcal{M}(2) \}$$

Mise en équation

Relation de récurrence

$$\mathcal{M}(k) = 1 + \min \{ \mathcal{M}(k - p) \mid p \in \text{pieces tq. } p \leq k \}$$

Cas de base / initialisation

$$\mathcal{M}(0) = 0$$

Algorithme naïf

Fonction $M(k, \text{pieces})$:

Si $k=0$ **Alors**

Renvoyer 0

Sinon

$L \leftarrow [M(k-p, \text{pieces}) \text{ pour } p \in \text{pieces tq.}$

$p \leq k]$

Renvoyer $1 + \min(L)$

Fin Fonction

Exercice

Consigne

Traduire ceci en code Python.

Test

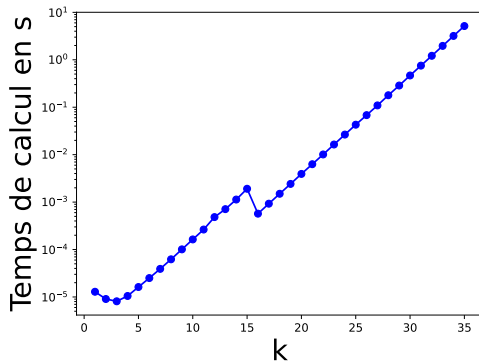
Vérifier que $\mathcal{M}(6) = 3$.

Test périlleux

Essayer de calculer $\mathcal{M}(50) = 3$.

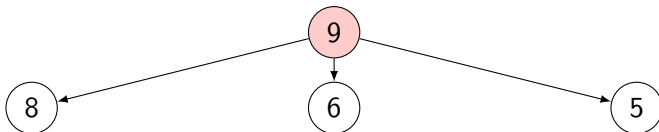
Complexité expérimentale

Avec $\text{pieces} = [1, 3, 4]$

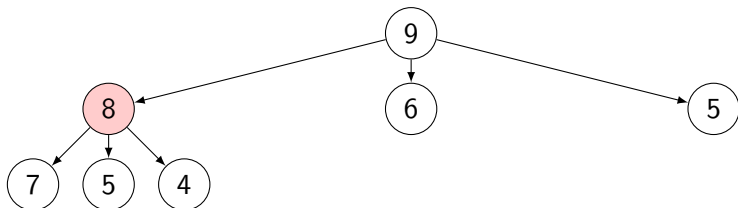


Complexité expérimentale du calcul naïf par récurrence

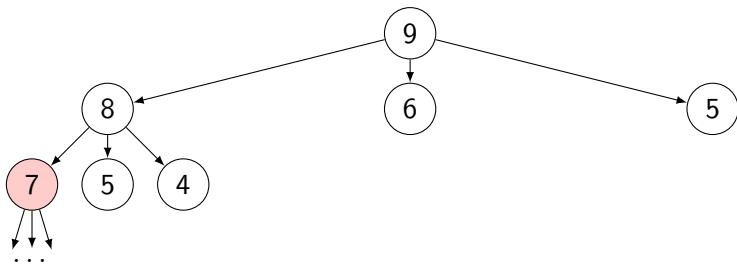
Premiers appels récursifs



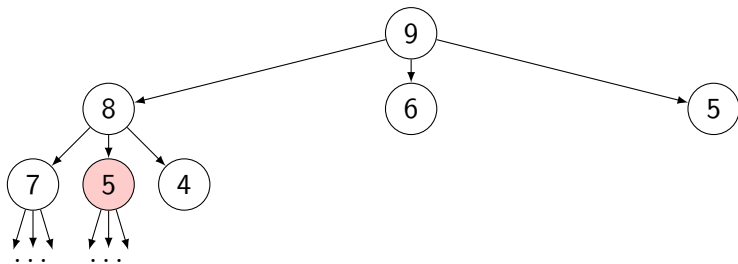
Premiers appels récurifs



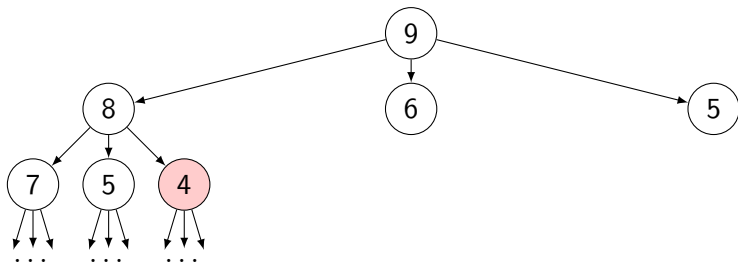
Premiers appels récursifs



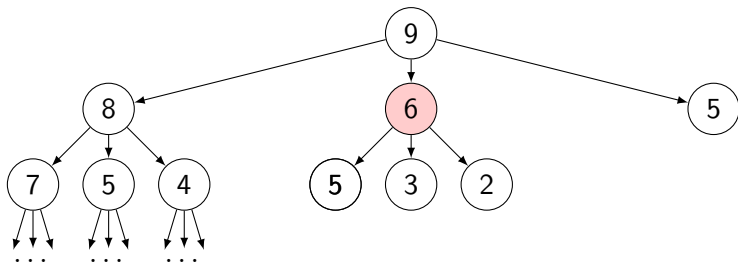
Premiers appels récursifs



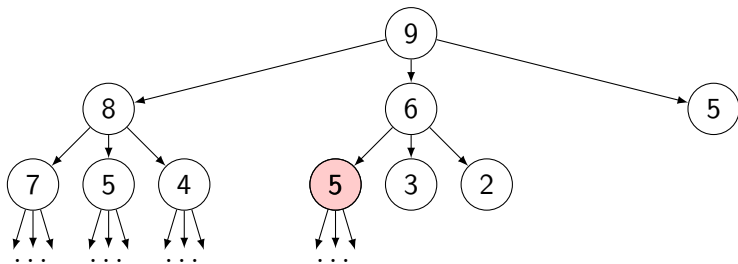
Premiers appels récursifs



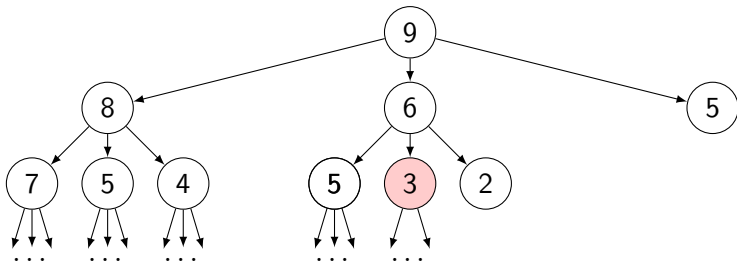
Premiers appels récurifs



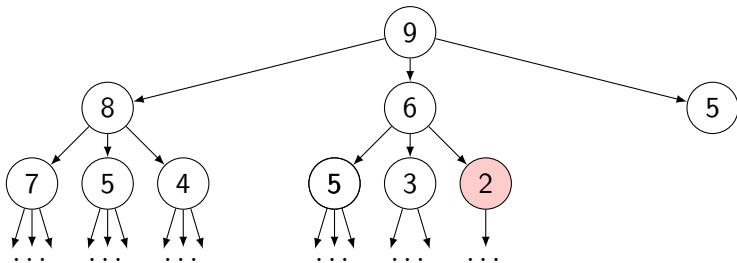
Premiers appels récursifs



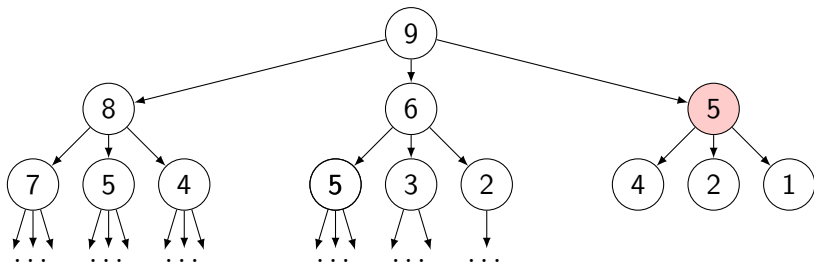
Premiers appels récursifs



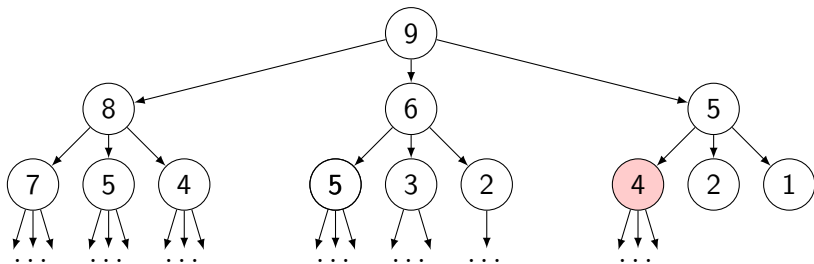
Premiers appels récursifs



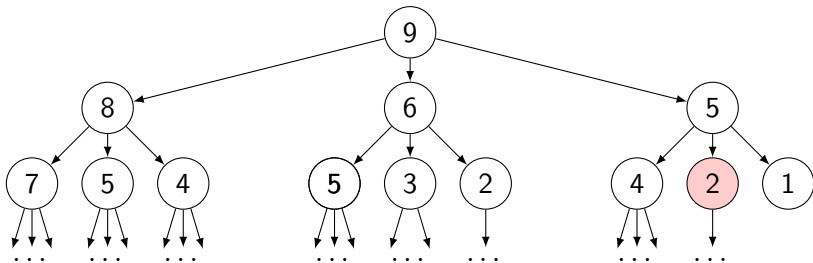
Premiers appels récursifs



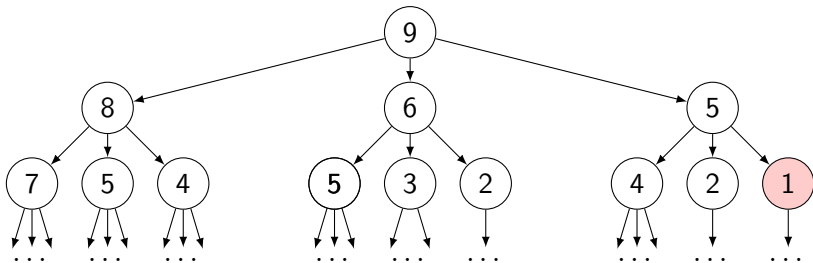
Premiers appels récursifs



Premiers appels récursifs



Premiers appels récursifs



Sur notre exemple

Nombre d'appels récursifs réalisés sur chaque valeur.

[40, 25, 15, 9, 6, 4, 2, 1, 1, 1]

Explication

On a calculé 15 fois $\mathcal{M}(2)$

Formalisation du problème

Le problème

- Ensemble d'objets O
- Fonction de coût $c : O \rightarrow \mathbb{R}_+$

Objectif(s) : optimiser

- Trouver $\min \{ c(o) \mid o \in O \}$
- Trouver $o^* \in O$ tel que $c(o^*) = \min \{ c(o) \mid o \in O \}$.

Principe de sous-structure optimale

Définition (sous-structure)

Une *sous-structure* est une restriction de notre problème à un ensemble plus petit.

Définition (sous-structure optimale)

Un problème vérifie la propriété de *sous-structure optimale* si une solution optimale de tout sous-problème est une partie d'une solution optimale de notre problème de départ.

Sur notre exemple

Avec $\text{pieces}=[1,3,4]$

Une solution optimale

- $18 = 4 + 4 + 4 + 3 + 3 = \underbrace{4 + 4 + 3} + \underbrace{4 + 3}$
- $\mathcal{M}(18) = 5$

Sous-problèmes

- $\mathcal{M}(11) = 3, 11 = 4 + 4 + 3$
- $\mathcal{M}(7) = 2, 7 = 4 + 3$
- On ne peut pas faire strictement mieux sur 11 et 7

Chevauchement de sous-problèmes

Définition (chevauchement de sous-problèmes)

Des sous-problèmes se *chevauchent* lorsque la résolution du problème appliqué à des sous-structures différentes amène à résoudre ce problème sur des mêmes sous-structures.

Conséquence concrète

On résout alors plusieurs fois le même sous-problème.

Sur notre exemple

Avec $\text{pieces} = [1, 3, 4]$

- Nombre d'appels pour chaque calcul

$[40, 25, 15, 9, 6, 4, 2, 1, 1, 1]$

- 15 calculs de $\mathcal{M}(1)$

Il y a bien chevauchement de sous-problèmes

Programmation dynamique

Définition (programmation dynamique)

Les méthodes de programmation dynamique sont mises en œuvre lorsque le problème étudié possède une propriété de sous-structure optimale, et lorsque les sous-problèmes utilisés pour le résoudre se chevauchent.

Équation de Bellman

Définition (équation de Bellman)

C'est la « relation de récurrence ».

Équation de Bellman de notre exemple

$$\mathcal{M}(k) = 1 + \min \{ \mathcal{M}(k - p) \mid p \in \text{pieces tq. } p \leq k \}$$

Cas d'initialisation de notre exemple

$$\mathcal{M}(0) = 0$$

Calcul récursif de haut en bas avec mémoïsation

Principe

- Même structure qu'une version récursive naïve.
- Sauvegarder/mémoïser les résultats dans une structure de données (initialement vide).
- Initialiser la structure de données par le cas d'initialisation.
- Ne pas recopier les données du problème : utiliser une sous-fonction auxiliaire.

Avant chaque calcul de sous-problème

- Vérifier d'abord si on l'a déjà calculé (alors : ne rien faire) ;
- sinon, on lance un calcul récursif.

Calcul de notre exemple, de haut en bas et avec mémoïsation

Fonction monnaieHautEnBas(*k*, *pieces*) :

 memo ← dictionnaire {0 : 0}

Fonction M(*i*) :

 ## *M(i)* renvoie $\mathcal{M}(i)$

Si $i \notin \text{memo}$ **Alors**

 memo[*i*]

 ← $1 + \min \{ M(i - p) \mid p \in \text{pieces tq. } p \leq i \}$

Renvoyer memo[*i*]

Fin Fonction

Renvoyer M(*k*)

Fin Fonction

Exercice

Consigne

Traduire ceci en code Python.

Tests pour `pieces=[1,3,4]`

- Calculer $\mathcal{M}(10)$.
- Calculer $\mathcal{M}(1000)$.

Étude de complexité

Complexité temporelle : modèle

- Opérations arithmétiques (addition) : $O(1)$;
- Minimum d'une liste de n entiers : $O(n)$

Complexité temporelle : calcul

- création de memo : $O(1)$
- un calcul à réaliser pour chaque entier $1 \leq i \leq k$
- pour chaque calcul, opérations élémentaires + minimum : $O(n)$

Complexité finale : $kO(n) = O(kn)$

Étude de complexité

Complexité spatiale

- coût de memo : $O(k)$
- coût du calcul de minimum : $O(n)$
- en fonction de comment est gérée la mémoire : $O(kn)$

Programmation impérative de bas en haut

Généralités

- Mise en œuvre en programmation impérative (boucles for)
- Souvent possible
- Souvent délicat à mettre en œuvre
- Parfois très efficace si bien mis en œuvre, notamment en complexité spatiale

Sur notre exemple

- Calcul de $\mathcal{M}(i)$: ne demande que des calculs de $\mathcal{M}(j)$ pour $j < i$.
- Calculer les $\mathcal{M}(i)$ à partir de $\mathcal{M}(0)$.

Exercise

Avec $\text{pieces} = [1, 3, 4]$, calculer $\mathcal{M}(10)$ en remplissant la grille suivante.

[illegible]

Programmation de bas en haut

Fonction monnaieBasEnHaut(k , pieces) :

```

| M ← [0]
| Pour i variant de 1 à k, Faire
|   | ajout de
1 + min({ M[i - p] | p ∈ pieces tq. p ≤ i }) à M
| Renvoyer M[k]

```

Fin Fonction

Exercice

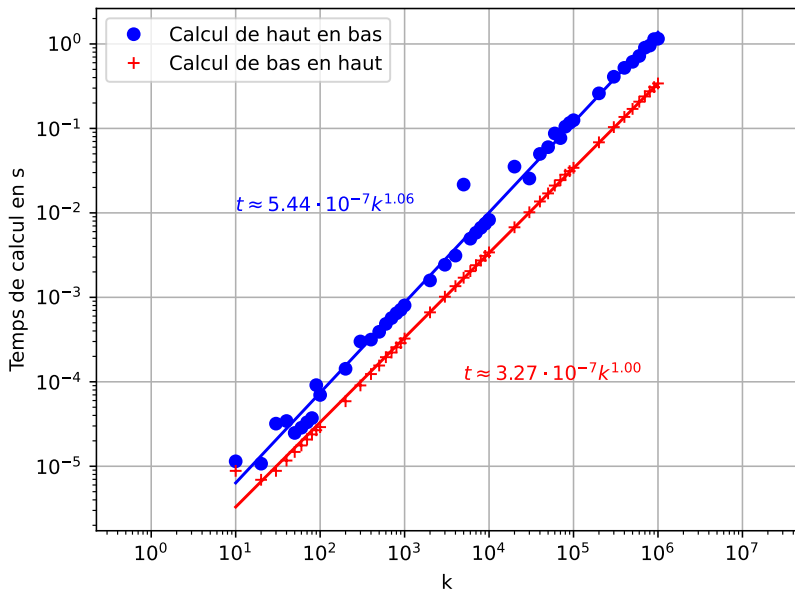
Consigne

Traduire ceci en code Python.

Tests pour `pieces=[1,3,4]`

- Calculer $\mathcal{M}(10)$.
- Calculer $\mathcal{M}(1000)$.

Comparaison des deux méthodes



Principe pour la version memoisée

Principe

Reconstruire une solution optimale o^* à partir de la structure mémoisée

Attention

Pas d'unicité de o^*

Sur notre exemple

Reconstruire la suite des modifications uniquement à partir du dictionnaire `memo`.

Étude de la relation de récurrence

Remarque fondamentale

Sont équivalentes :

- p apparaît dans une décomposition optimale de i
- $\mathcal{M}(i - p) = 1 + \min \{ \mathcal{M}(i - p') \mid p' \in \text{pieces tq. } p' \leq i \}$
- $\mathcal{M}(i - p) = 1 + \mathcal{M}(i)$

Conclusion

« Remonter » vers les cases les plus petites.

Exercice

On donne la liste des $\mathcal{M}(i)$, pour le système de pièces $[1, 3, 4]$.

i	0	1	2	3	4	5	6	7	8	9	10
$\mathcal{M}(i)$	0	1	2	1	1	2	2	2	2	3	3

Déterminer une manière optimale de décomposer 10 sur ce système.

Sur notre exemple

Fonction reconstruction(k, memo, pieces):

Si k=0 **Alors**

Renvoyer []

Sinon

Pour chaque p **De** pieces, **Faire**

Si $p \leq k$ et $\text{memo}[k-p]+1 = \text{memo}[p]$ **Alors**

 L $\leftarrow$ reconstruction(k-p, memo, pieces)

 ajouter p à L

Renvoyer L

Fin Fonction

Version itérative (de bas en haut)

- Construire la solution optimale pas à pas.
- Nécessite d'adapter la structure de données : conserver simultanément
 - la valeur du minimum ;
 - une solution optimale.
- Souvent lourd

Exemple

Exercice

Avec $\text{pieces} = [1, 3, 4]$, calculer $\mathcal{M}(10)$ et une décomposition optimale de 10 en remplissant la grille suivante.

i	0	1	2	3	4	5	6	7	8	9	10
$\mathcal{M}(i)$	0										
Déc.	[]										

Sur notre exemple (1)

- M : liste des (i, l)
- I : liste, décomposition optimale de i

Fonction meilleurePiece(i , M , pieces) :

$L \leftarrow [M[i-p][0] \text{ pour } p \in \text{pieces tq. } p \leq i]$

$\text{mini} \leftarrow \min(L)$

Pour chaque p **De** pieces, **Faire**

Si $p \leq i$ et $M[i-p][0] = \text{mini}$ **Alors**

$\text{dec} \leftarrow \text{copie de } M[i-p][1]$

ajouter p à dec

Renvoyer ($\text{mini}+1$, dec)

Fin Fonction

Sur notre exemple (2)

Fonction monnaieBasEnHaut2(k, pieces) :

 | M $\leftarrow$ [(0, [])]

 | **Pour** i variant de 1 à k, **Faire**

 | ajouter meilleurePiece(i, M, pieces) à M

 | **Renvoyer** M[k]

Fin Fonction

Suite de Fibonacci

Définition

$$F_0 = 0, \quad F_1 = 1, \quad \forall n \geq 2, \quad F_n = F_{n-1} + F_{n-2}.$$

Instruction

Écrire un algorithme de calcul de cette suite de Fibonacci par programmation dynamique, de haut en bas.

Le problème d'ordonnancement de tâches

Données du problème

Tâches $T_0, \dots, T_{n-1}$ décrites par :

- dates de début $d_0, \dots, d_{n-1}$;
- dates de fin $f_0 \leq \dots \leq f_{n-1}$;
- poids $p_0, \dots, p_{n-1}$.

Si $i < j$, T_i et T_j sont compatibles si $f_i \leq d_j$.

Objectif

Trouver l'ordonnancement de tâches compatibles de poids maximal (ou juste ce poids).

Équation de Bellman

Notations

- S_i : poids maximal des tâches compatibles parmi $T_0, \dots, T_i$
- j est le plus grand entier vérifiant $f_j \leq d_{i+1}$ (si aucune tâche n'est terminée, $S_j = 0$)

Cas d'initialisation

$$S_0 = p_0$$

Équation de Bellman

$$S_{i+1} = \max(S_j + p_{i+1}, S_i).$$

Calcul de bas en haut

Fonction ordonnancement(d, f, p) :

$n \leftarrow$ nombre de tâches

$S \leftarrow$ tableau de longueur n contenant des $p[0]$

Pour i variant de 1 à $n-1$, **Faire**

$j \leftarrow$ plus grand $k < i$ tel que $f[k] \leq d[i]$ (-1
s'il n'y en a pas)

$S[i] \leftarrow \max(S[i-1], S[j] + p[i])$ avec la
convention $S[-1] = 0$

Renvoyer $S[n-1]$

Fin Fonction

Notre exemple

n° de tâche i	début (d_i)	fin (f_i)	poids (p_i)
0	0	2	3
1	3	4	1
2	2	4	2
3	1	5	6
4	2	5	1
5	6	7	1
6	4	8	2
7	4	9	2
8	7	10	3
9	3	10	1
10	6	10	3

Plus long sous-tableau croissant

Définition

Si $T = [t_0, \dots, t_{n-1}]$, alors un sous-tableau de T est un tableau extrait de T , i.e. un tableau $T' = [t_{i_0}, \dots, t_{i_{p-1}}]$, où $0 \leq i_0 < \dots < t_{i_{p-1}} < n$.

Problème

Étant donné un tel tableau T d'entiers, quelle est la longueur maximal d'un tel sous-tableau T' qui est strictement croissant ?
On aimerait aussi donner un tel sous-tableau T' .

Questions (1)

- Combien y-a-t-il de sous-tableaux T' de T , en fonction de n ?
Quelle serait la complexité temporelle d'une recherche exhaustive de la longueur du plus grand sous-tableau croissant de T ?
- Si $0 \leq j < n$, on considère le tableau $T' = [t_{i_0}, \dots, t_{i_{p-2}}, t_j]$ ayant la propriété suivante : « T' est le plus grand sous-tableau de T strictement croissant et dont le dernier élément est t_j ». Que peut-on dire de $T'' = [t_{i_0}, \dots, t_{i_p}]$?
- On note ℓ_j la longueur de ce sous-tableau T' . Écrire une relation liant ℓ_j à $\ell_0, \dots, \ell_{j-1}$. On remarquera que cette relation devra faire apparaître les conditions « $t_i < t_j$ » pour $0 \leq i < j$.
- Que vaut ℓ_0 ?

Questions (2)

- En déduire un pseudo-code permettant de calculer $L = [\ell_0, \dots, \ell_{n-1}]$.
- À partir de ce tableau L , comment peut-on répondre au problème initial ?
- Évaluer la complexité temporelle de l'algorithme ainsi construit.