

Programmation dynamique

Quentin Fortier

September 22, 2021

Sous-problèmes

Souvent, un problème peut se ramener à l'étude de sous-problèmes (le même problème, mais en plus petit).

Sous-problèmes

Souvent, un problème peut se ramener à l'étude de sous-problèmes (le même problème, mais en plus petit). Exemple pour le calcul des termes de la suite de Fibonacci :

$$u_0 = 1$$

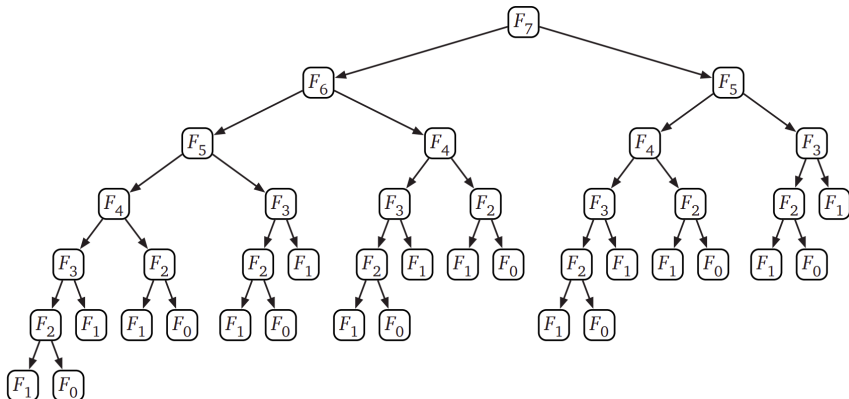
$$u_1 = 1$$

$$u_n = u_{n-1} + u_{n-2}$$

```
def fibo(n):  
    if n <= 1:  
        return 1  
    return fibo(n - 1) + fibo(n - 2)
```

Sous-problèmes

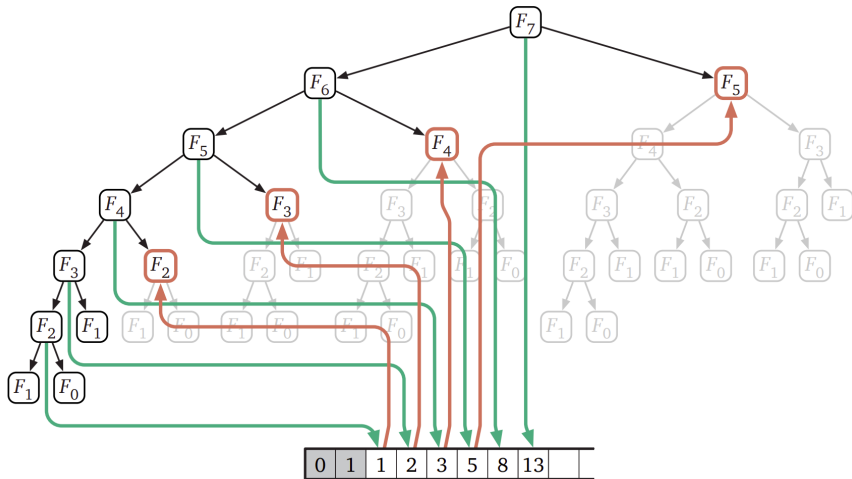
Problème : le même sous-problème est résolu plusieurs fois, ce qui est inutile et inefficace.



Idée : stocker les valeurs des sous-problèmes pour éviter de les calculer plusieurs fois.

```
def fibo(n):  
    F = [1, 1] # F[n] va contenir le nème terme  
    for i in range(n - 1):  
        F.append(F[-1] + F[-2])  
    return F[-1]
```

Sous-problèmes



Dans le cas de la suite de Fibonacci, on peut mémoriser seulement les 2 derniers termes :

```
def fibo(n):  
    f0, f1 = 1, 1  
    for i in range(n - 1):  
        f0, f1 = f1, f0 + f1  
    return f1
```

Pour résoudre un problème de programmation dynamique :

- 1 Chercher une équation de récurrence. Souvent, cela demande d'introduire un paramètre.

Pour résoudre un problème de programmation dynamique :

- ❶ Chercher une équation de récurrence. Souvent, cela demande d'introduire un paramètre.
- ❷ Stocker en mémoire les résultats des sous-problèmes pour éviter de les calculer plusieurs fois.

Sac à dos

Problème (sac à dos)

Entrée : un sac à dos de capacité C , des objets o_1, \dots, o_n de poids p_1, \dots, p_n et valeurs v_1, \dots, v_n .

Sortie : la valeur maximum que l'on peut mettre dans le sac.

Problème (sac à dos)

Entrée : un sac à dos de capacité C , des objets o_1, \dots, o_n de poids p_1, \dots, p_n et valeurs v_1, \dots, v_n .

Sortie : la valeur maximum que l'on peut mettre dans le sac.

Soit $v[c][k]$ la valeur maximum que l'on peut mettre dans un sac de capacité c , en ne considérant que les objets o_1, \dots, o_k .

Problème (sac à dos)

Entrée : un sac à dos de capacité de capacité C , des objets o_1, \dots, o_n de poids p_1, \dots, p_n et valeurs v_1, \dots, v_n .

Sortie : la valeur maximum que l'on peut mettre dans le sac.

Soit $v[c][k]$ la valeur maximum que l'on peut mettre dans un sac de capacité c , en ne considérant que les objets o_1, \dots, o_k .

$$v[c][0] = 0, \quad \forall 0 \leq c \leq C$$

$$v[c][k] = \max(v[c][k-1], v[c-p_k][k-1] + v_k), \quad \forall 0 \leq c \leq C$$

Sac à dos

Problème (sac à dos)

Entrée : un sac à dos de capacité de capacité C , des objets o_1, \dots, o_n de poids p_1, \dots, p_n et valeurs v_1, \dots, v_n .

Sortie : la valeur maximum que l'on peut mettre dans le sac.

Soit $v[c][k]$ la valeur maximum que l'on peut mettre dans un sac de capacité c , en ne considérant que les objets o_1, \dots, o_k .

$$v[c][0] = 0, \quad \forall 0 \leq c \leq C$$

$$v[c][k] = \max(\underbrace{v[c][k-1]}_{\text{sans prendre } o_k}, \underbrace{v[c-p_k][k-1] + v_k}_{\text{en prenant } o_k}), \quad \forall 0 \leq c \leq C$$

Résolution du sac à dos par programmation dynamique

Pour $c = 0$ à C :

$$v[c][0] \leftarrow 0$$

Pour $k = 1$ à n :

Pour $c = 0$ à C :

$$v[c][k] \leftarrow \max(v[c][k-1], v[c-p_k][k-1] + v_k)$$

Complexité :

Résolution du sac à dos par programmation dynamique

Pour $c = 0$ à C :

$$v[c][0] \leftarrow 0$$

Pour $k = 1$ à n :

Pour $c = 0$ à C :

$$v[c][k] \leftarrow \max(v[c][k-1], v[c-p_k][k-1] + v_k)$$

Complexité : $O(nC)$

Comme on a juste besoin de stocker $v[\dots][k - 1]$ pour calculer $v[\dots][k]$:

Comme on a juste besoin de stocker $v[\dots][k - 1]$ pour calculer $v[\dots][k]$:

Résolution du sac à dos par programmation dynamique

Pour $c = 0$ à C :

$v[c] \leftarrow 0$ Pour $k = 1$ à n :

Pour $c = 0$ à C :

$v[c] \leftarrow \max(v[c], v[c - p_k] + v_k)$

L'algorithme de Bellman-Ford permet de résoudre le problème suivant :

Problème (plus courts chemins)

Entrée : $G = (V, E)$ un graphe orienté pondéré sans cycle de poids négatif et $r \in V$.

L'algorithme de Bellman-Ford permet de résoudre le problème suivant :

Problème (plus courts chemins)

Entrée : $G = (V, E)$ un graphe orienté pondéré sans cycle de poids négatif et $r \in V$.

Sortie : un tableau T tel que si $v \in V$, $T[v]$ contient la distance (= longueur d'un plus court chemin) de r à v .

Soit $d_k(v)$ le poids minimum d'un chemin de r à v utilisant au plus k arêtes.

Soit $d_k(v)$ le poids minimum d'un chemin de r à v utilisant au plus k arêtes.

Soit $d_k(v)$ le poids minimum d'un chemin de r à v utilisant au plus k arêtes.

$$d_{k+1}(v) = \min_{(u,v) \in E} d_k(u) + w(u, v)$$

Soit $d_k(v)$ le poids minimum d'un chemin de r à v utilisant au plus k arêtes.

$$d_{k+1}(v) = \min_{(u,v) \in E} d_k(u) + w(u, v)$$

Preuve : soit C un plus court chemin de r à v utilisant au plus $k + 1$ arêtes.

Soit $d_k(v)$ le poids minimum d'un chemin de r à v utilisant au plus k arêtes.

$$d_{k+1}(v) = \min_{(u,v) \in E} d_k(u) + w(u, v)$$

Preuve : soit C un plus court chemin de r à v utilisant au plus $k + 1$ arêtes.

Soit u le prédécesseur de v dans C .

Soit $d_k(v)$ le poids minimum d'un chemin de r à v utilisant au plus k arêtes.

$$d_{k+1}(v) = \min_{(u,v) \in E} d_k(u) + w(u, v)$$

Preuve : soit C un plus court chemin de r à v utilisant au plus $k + 1$ arêtes.

Soit u le prédécesseur de v dans C .

Alors le sous-chemin de C de r à u est un plus court chemin utilisant au plus k arêtes

Soit $d_k(v)$ le poids minimum d'un chemin de r à v utilisant au plus k arêtes.

$$d_{k+1}(v) = \min_{(u,v) \in E} d_k(u) + w(u, v)$$

Preuve : soit C un plus court chemin de r à v utilisant au plus $k + 1$ arêtes.

Soit u le prédecesseur de v dans C .

Alors le sous-chemin de C de r à u est un plus court chemin utilisant au plus k arêtes (s'il y avait un chemin plus court que C' , on pourrait le remplacer dans C ce qui contredirait la minimalité de C).

Soit $d_k(v)$ le poids minimum d'un chemin de r à v utilisant au plus k arêtes.

$$d_{k+1}(v) = \min_{(u,v) \in E} d_k(u) + w(u, v)$$

Preuve : soit C un plus court chemin de r à v utilisant au plus $k + 1$ arêtes.

Soit u le prédecesseur de v dans C .

Alors le sous-chemin de C de r à u est un plus court chemin utilisant au plus k arêtes (s'il y avait un chemin plus court que C' , on pourrait le remplacer dans C ce qui contredirait la minimalité de C).

Remarque C' est une propriété de **sous-structure optimale** : un sous-chemin d'un plus court chemin est aussi un plus court chemin.

Bellman-Ford

On va utiliser un tableau $d[v][k]$ pour stocker $d_k(v)$.

Algorithme de Bellman-Ford

Initialiser $d[r] \leftarrow 0$ et $d[v] \leftarrow \infty, \forall k, \forall v \neq r$

Pour $k = 0$ à $|V| - 2$:

 Pour tout sommet v :

 Pour tout arc (u, v) rentrant dans v :

 Si $d[u][k] + w(u, v) < d[v][k + 1]$:

$d[v][k + 1] \leftarrow d[u][k] + w(u, v)$

Comme on a juste besoin de stocker $d[...][k - 1]$ pour calculer $d[...][k]$:

Bellman-Ford

Comme on a juste besoin de stocker $d[\dots][k-1]$ pour calculer $d[\dots][k]$:

Algorithme de Bellman-Ford

Initialiser $d[r][0] \leftarrow 0$ et $d[v][0] \leftarrow \infty, \forall v \neq r$

Pour $k = 0$ à $|V| - 2$:

 Pour tout sommet v :

 Pour tout arc (u, v) rentrant dans v :

 Si $d[u] + w(u, v) < d[v]$:

$d[v] \leftarrow d[u] + w(u, v)$

Sous-suite croissante

Soit T un tableau.

Définition

Une **sous-suite croissante** de T correspond à des éléments $T[i_1] \leq T[i_2] \leq \dots \leq T[i_k]$ avec $i_1 \leq i_2 \leq \dots \leq i_k$.

Sous-suite croissante

Soit T un tableau.

Définition

Une **sous-suite croissante** de T correspond à des éléments $T[i_1] \leq T[i_2] \leq \dots \leq T[i_k]$ avec $i_1 \leq i_2 \leq \dots \leq i_k$.

Problème

Trouver la longueur maximum d'une sous-suite croissante de T .

Sous-suite croissante

Soit T un tableau.

Définition

Une **sous-suite croissante** de T correspond à des éléments $T[i_1] \leq T[i_2] \leq \dots \leq T[i_k]$ avec $i_1 \leq i_2 \leq \dots \leq i_k$.

Problème

Trouver la longueur maximum d'une sous-suite croissante de T .

Exemple :

$$T = [8, 1, 3, 7, 5, 6, 4]$$

Sous-suite croissante

Soit T un tableau.

Définition

Une **sous-suite croissante** de T correspond à des éléments $T[i_1] \leq T[i_2] \leq \dots \leq T[i_k]$ avec $i_1 \leq i_2 \leq \dots \leq i_k$.

Sous-suite croissante

Soit T un tableau.

Définition

Une **sous-suite croissante** de T correspond à des éléments $T[i_1] \leq T[i_2] \leq \dots \leq T[i_k]$ avec $i_1 \leq i_2 \leq \dots \leq i_k$.

Problème

Trouver la longueur maximum d'une sous-suite croissante de T .

Sous-suite croissante

Soit T un tableau.

Définition

Une **sous-suite croissante** de T correspond à des éléments $T[i_1] \leq T[i_2] \leq \dots \leq T[i_k]$ avec $i_1 \leq i_2 \leq \dots \leq i_k$.

Problème

Trouver la longueur maximum d'une sous-suite croissante de T .

Exemple :

$$T = [8, 1, 3, 7, 5, 6, 4]$$

Longueur maximum : 4.

Sous-suite croissante

Soit T un tableau.

Soit $L[k]$ la longueur d'une **plus longue sous-suite croissante (PLSSC)** terminant en $T[k]$ (c'est à dire de la forme $T[i_1] \leq T[i_2] \leq \dots \leq T[i_p] = T[k]$).

Sous-suite croissante

Soit T un tableau.

Soit $L[k]$ la longueur d'une **plus longue sous-suite croissante (PLSSC)** terminant en $T[k]$ (c'est à dire de la forme $T[i_1] \leq T[i_2] \leq \dots \leq T[i_p] = T[k]$).

Exemple :

$$T = [8, 1, 3, 7, 5, 6, 4]$$

PLSSC terminant en $T[6]$ ($= 4$) :

Sous-suite croissante

Soit T un tableau.

Soit $L[k]$ la longueur d'une **plus longue sous-suite croissante (PLSSC)** terminant en $T[k]$ (c'est à dire de la forme $T[i_1] \leq T[i_2] \leq \dots \leq T[i_p] = T[k]$).

Exemple :

$$T = [8, 1, 3, 7, 5, 6, 4]$$

PLSSC terminant en $T[6]$ ($= 4$) :

$$T = [8, \textcolor{red}{1}, \textcolor{red}{3}, 7, 5, 6, \textcolor{red}{4}]$$

$$L[6] = 3$$

Sous-suite croissante

Soit $T[i_1] \leq \dots \leq T[i_{p-1}] \leq T[i_p] = T[k]$ une PLSSC terminant en $T[k]$.

Sous-suite croissante

Soit $T[i_1] \leq \dots \leq T[i_{p-1}] \leq T[i_p] = T[k]$ une PLSSC terminant en $T[k]$.

Alors $T[i_1] \leq \dots \leq T[i_{p-1}]$ est une PLSSC terminant en $T[i_{p-1}]$

Sous-suite croissante

Soit $T[i_1] \leq \dots \leq T[i_{p-1}] \leq T[i_p] = T[k]$ une PLSSC terminant en $T[k]$.

Alors $T[i_1] \leq \dots \leq T[i_{p-1}]$ est une PLSSC terminant en $T[i_{p-1}]$ (s'il y avait une PLSSC plus grande on pourrait l'utiliser dans la PLSSC initiale pour contredire sa maximalité).

Sous-suite croissante

Soit $T[i_1] \leq \dots \leq T[i_{p-1}] \leq T[i_p] = T[k]$ une PLSSC terminant en $T[k]$.

Alors $T[i_1] \leq \dots \leq T[i_{p-1}]$ est une PLSSC terminant en $T[i_{p-1}]$ (s'il y avait une PLSSC plus grande on pourrait l'utiliser dans la PLSSC initiale pour contredire sa maximalité).

Donc :

$$L[k] = 1 + L[i_{p-1}]$$

Sous-suite croissante

Soit $T[i_1] \leq \dots \leq T[i_{p-1}] \leq T[i_p] = T[k]$ une PLSSC terminant en $T[k]$.

Alors $T[i_1] \leq \dots \leq T[i_{p-1}]$ est une PLSSC terminant en $T[i_{p-1}]$ (s'il y avait une PLSSC plus grande on pourrait l'utiliser dans la PLSSC initiale pour contredire sa maximalité).

Donc :

$$L[k] = 1 + L[i_{p-1}]$$

Comme on ne connaît pas i_{p-1} , on peut essayer toutes les possibilités et conserver le maximum :

$$L[k] = 1 + \max_{\substack{i \leq k \\ T[i] \leq T[k]}} L[i]$$