Programmation dynamique

Quentin Fortier

September 21, 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 :

$$u_0 = 1$$

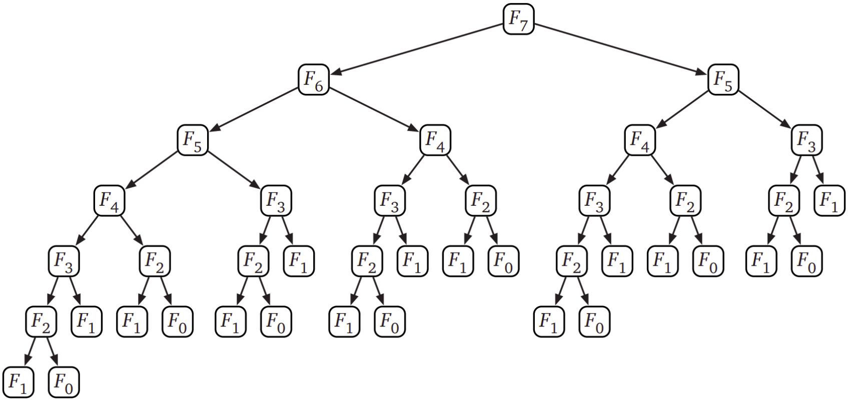
$$u_1 = 1$$

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

```python
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.

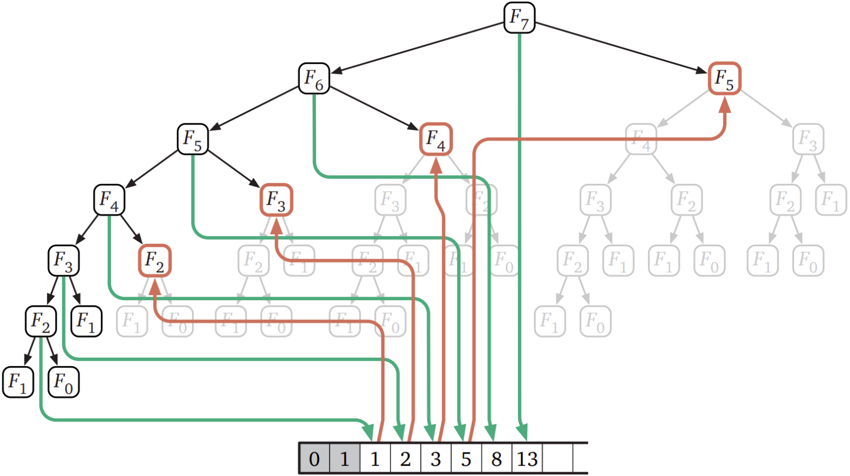


Sous-problèmes

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



Sous-problèmes

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

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

Programmation dynamique

Pour résoudre un problème de programmation dynamique, on commence par chercher une équation de récurrence.

Programmation dynamique

Pour résoudre un problème de programmation dynamique, on commence par chercher une équation de récurrence.
Souvent, cela demande d'introduire un paramètre.

Puis on stocke 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 de capacité C, des objets $o_1, ..., o_n$ de poids $p_1, ..., p_n$ et valeurs v_1, ..., v_n.
Sortie : la valeur maximum que l'on peut mettre dans le sac.

Sac à dos

Problème (sac à dos)

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

Sac à dos

Problème (sac à dos)

Entrée : un sac à dos de capacité de capacité C, des objets $o_1, \ldots, o_n$ de poids $p_1, \ldots, p_n$ et valeurs v_1, …, 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, …, 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 $o_1, ..., o_n$ de poids $p_1, ..., p_n$ et valeurs v_1, ..., 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, ..., o_k.

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

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

Sac à dos

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 :

Sac à dos

Résolution du sac à dos par programmation dynamique

Pour $c = 0$ à C:

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

Pour $k = 1$ à n:

$\quad$ Pour $c = 0$ à C:

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

Complexité : $O(nC)$

Sac à dos

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

Sac à dos

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

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

Bellman-Ford

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 $r \in V$.

Bellman-Ford

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.

Bellman-Ford

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

Bellman-Ford

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

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

Bellman-Ford

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

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

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

Bellman-Ford

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

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

<u>Preuve</u> : soit C un plus court chemin de r à v utilisant au plus $k + 1$ arêtes.
Soit u le prédecesseur de v dans C.

Bellman-Ford

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

Bellman-Ford

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).

Bellman-Ford

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

Algorithme de Bellman-Ford

Initialiser `d[r][0] <- 0` et `d[v][k] <-` ∞, $\forall$ 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)`

Bellman-Ford

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

Bellman-Ford

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

Algorithme de Bellman-Ford

Initialiser `d[0][r] <- 0` et `d[0][v] <-` ∞, $\forall$ `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)`