Programmation dynamique

Quentin Fortier

September 29, 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 :

$$u_0 = 0$$

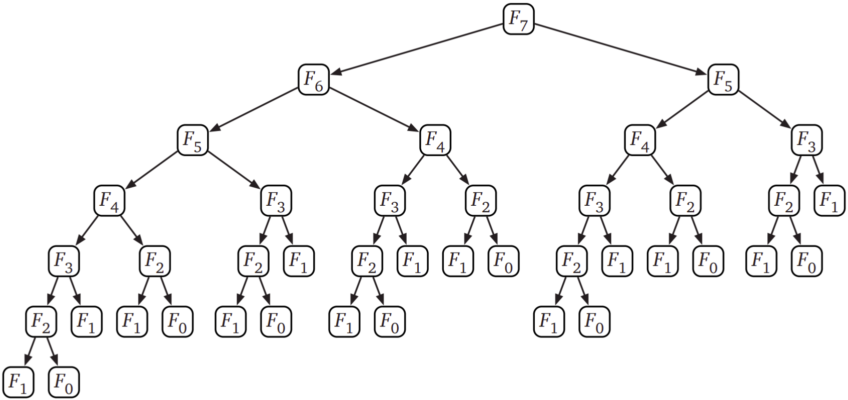
$$u_1 = 1$$

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

```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.

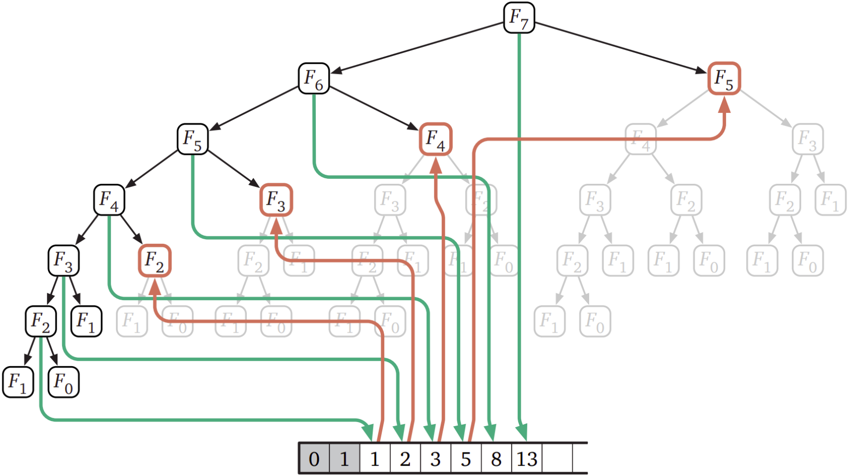


Sous-problèmes

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

```
def fibo(n):
    F = [0, 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 = 0, 1
    for i in range(n - 1):
        f0, f1 = f1, f0 + f1
    return f1
```

Programmation dynamique

Pour résoudre un problème de programmation dynamique :

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

Programmation dynamique

Pour résoudre un problème de programmation dynamique :

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

Programmation dynamique

Nous allons voir 3 exemples d'applications :

- Sac à dos
- Bellman-Ford pour trouver des plus courts chemins dans un graphe
- Trouver une sous-suite croissante maximale dans un tableau

Sac à dos

Problème (sac à dos)

Entrée : un sac à dos 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 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.

Sac à dos

Problème (sac à dos)

Entrée : un sac à dos 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.

$$v[c][0] = 0$$

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

Sac à dos

Problème (sac à dos)

Entrée : un sac à dos 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.

$$v[c][0] = 0$$

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

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:
        Si $c - p_k \geq 0$:
            $v[c][k] \leftarrow \max(v[c][k-1], v[c-p_k][k-1] + v_k)$
        Sinon:
            $v[c][k] \leftarrow v[c][k-1]$

Complexité :

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:
        Si $c - p_k \geq 0$:
            $v[c][k] \leftarrow \max(v[c][k-1], v[c-p_k][k-1] + v_k)$
        Sinon:
            $v[c][k] \leftarrow v[c][k-1]$

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:
  $v' \leftarrow$ copie de v
  Pour $c = 0$ à C:
    Si $c - p_k \geq 0$:
      $v'[c] \leftarrow \max(v[c], v[c - p_k] + v_k)$
  $v \leftarrow v'$

Mémoïsation

La programmation dynamique est une stratégie **bottom-up** : on résout les problèmes du plus petit au plus grand. On utilise des boucles for.

Mémoïsation

La programmation dynamique est une stratégie **bottom-up** : on résout les problèmes du plus petit au plus grand. On utilise des boucles for.
La **mémoîsation** est similaire mais avec une stratégie **top-down** : on

part du problème initial pour le décomposer. On utilise des appels récursifs.

Mémoïsation

La programmation dynamique est une stratégie **bottom-up** : on résout les problèmes du plus petit au plus grand. On utilise des boucles for.
La **mémoîsation** est similaire mais avec une stratégie **top-down** : on

part du problème initial pour le décomposer. On utilise des appels récursifs.
Pour éviter de résoudre plusieurs fois le même problème (comme pour Fibonacci), on mémorise (dans un tableau ou une table de hachage) les arguments pour lesquelles la fonction récursive a déjà été calculée.

Mémoïsation

Exemple de mémoïsation avec Fibonacci :

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def fibo(n):
    if n <= 1:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

`lru_cache` permet de se souvenir des appels déjà effectués et éviter de faire plusieurs fois le même appel récursif.

Mémoïsation

Exemple de mémoïsation avec Fibonacci :

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibo(n):
    if n <= 1:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

`lru_cache` permet de se souvenir des appels déjà effectués et éviter de faire plusieurs fois le même appel récursif.

Exercice : résoudre le sac à dos avec mémoïsation.

Bellman-Ford

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

Problème (plus courts chemins)

Entrée : $G = (V, \vec{E})$ un graphe orienté pondéré par w 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.

Bellman-Ford

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

Problème (plus courts chemins)

Entrée : $G = (V, \vec{E})$ un graphe orienté pondéré par w 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.

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

Preuve : 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.

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

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.

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

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

`d[r] <- 0`
Pour `v` $\neq$ `r`:
    Pour `k` $= 0$ à $|V| - 2$:
        `d[v][k] <-` ∞

Pour $k = 0$ à $|V| - 2$:
    Pour tout sommet `v`:
        Pour tout arc (`u`, `v`) entrant 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

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

Algorithme de Bellman-Ford

d[r] <- 0
Pour v $\neq$ r:
  Pour k $= 0$ à $|V| - 2$:
    d[v][k] <- ∞

Pour $k = 0$ à $|V| - 2$:
  Pour tout sommet v:
    Pour tout arc (u, v) entrant dans v:
      Si d[u][k] + w(u, v) < d[v][k + 1]:
        d[v][k + 1] $\leftarrow$ d[u][k] + w(u, v)

Complexité :

Bellman-Ford

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

Algorithme de Bellman-Ford

```
d[r] <- 0
Pour v ≠ r:
    Pour k = 0 à |V| − 2:
        d[v][k] <- ∞

Pour k = 0 à |V| − 2:
    Pour tout sommet v:
        Pour tout arc (u, v) entrant dans v:
            Si d[u][k] + w(u, v) < d[v][k + 1]:
                d[v][k + 1] ← d[u][k] + w(u, v)
```

<u>Complexité</u> : $O(np)$ où $n = |V|$ et $p = |\vec{E}|$.

Bellman-Ford

Parcourir tous les sommets puis tous les arcs (`u`, `v`) entrants dans `v` revient à parcourir tous les arcs du graphe :

Algorithme de Bellman-Ford

```
d[r] <- 0
Pour v ≠ r:
    Pour k = 0 à |V| − 2:
        d[v][k] <- ∞

Pour k = 0 à |V| − 2:
    Pour tout arc (u, v):
        Si d[u][k] + w(u, v) < d[v][k + 1]:
            d[v][k + 1] ← d[u][k] + w(u, v)
```

Bellman-Ford

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

Bellman-Ford

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

Algorithme de Bellman-Ford

d[r] <- 0
Pour v $\neq$ r:
    d[v] <- ∞

Pour $k = 0$ à $|V| - 2$:
    d' $\leftarrow$ copie de d
    Pour tout arc (u, v):
        Si d[u] + w(u, v) < d[v]:
            d'[v] $\leftarrow$ d[u] + w(u, v)
    d $\leftarrow$ d'

Sous-suite croissante

Soit T un tableau.

Définition

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

Sous-suite croissante

Soit T un tableau.

Définition

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

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 $T[i_1] \leq T[i_2] \leq ... \leq T[i_k]$ avec $i_1 \leq i_2 \leq ... \leq i_k$.

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 $T[i_1] \leq T[i_2] \leq ... \leq T[i_k]$ avec $i_1 \leq i_2 \leq ... \leq i_k$.

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 (**LIS** en anglais, pour Longest Increasing Subsequence) terminant en $T[k]$ (c'est à dire de la forme $T[i_1] \leq T[i_2] \leq ... \leq T[i_p] = T[k]$).

Sous-suite croissante

Soit T un tableau.
Soit $L[k]$ la longueur d'une plus longue sous-suite croissante (**LIS** en anglais, pour Longest Increasing Subsequence) terminant en $T[k]$ (c'est à dire de la forme $T[i_1] \leq T[i_2] \leq ... \leq T[i_p] = T[k]$).

Exemple :

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

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

Sous-suite croissante

Soit T un tableau.
Soit $L[k]$ la longueur d'une plus longue sous-suite croissante (**LIS** en anglais, pour Longest Increasing Subsequence) terminant en $T[k]$ (c'est à dire de la forme $T[i_1] \leq T[i_2] \leq ... \leq T[i_p] = T[k]$).

Exemple :

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

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

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

$L[6] = 3$

Sous-suite croissante

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

Sous-suite croissante

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

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

Sous-suite croissante

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

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

Sous-suite croissante

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

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

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

Sous-suite croissante

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

Alors $T[i_1] \leq ... \leq T[i_{p-1}]$ est une LIS terminant en $T[i_{p-1}]$ (s'il y avait une LIS plus grande on pourrait l'utiliser dans la LIS 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 :

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

Annexe : Théorème d'Erdős-Szekeres

Lemme

Supposons que $L[k]$ contienne p fois la même valeur. Montrer que T possède une sous-suite décroissante de longueur p.

Annexe : Théorème d'Erdős-Szekeres

Lemme

Supposons que $L[k]$ contienne p fois la même valeur. Montrer que T possède une sous-suite décroissante de longueur p.

Théorème d'Erdős-Szekeres

Si n est la taille de T, montrer que T contient soit une sous-suite croissante de longueur $\lfloor \sqrt{n} \rfloor$, soit une sous-suite décroissante de longueur $\lfloor \sqrt{n} \rfloor$