Programmation dynamique

Thibaut Cantaluppi

September 26, 2024

Sous-problèmes

Pour résoudre un problème, il est courant de le ramener à des sous-problèmes plus simples.

Sous-problèmes

Pour résoudre un problème, il est courant de le ramener à des sous-problèmes plus simples.

Deux grandes méthodes pour le faire :

1. **Diviser pour régner** : résoudre des sous-problèmes **indépendants** (récursivement) puis les combiner pour obtenir une solution du problème initial.

Sous-problèmes

Pour résoudre un problème, il est courant de le ramener à des sous-problèmes plus simples.

Deux grandes méthodes pour le faire :

1. **Diviser pour régner** : résoudre des sous-problèmes **indépendants** (récursivement) puis les combiner pour obtenir une solution du problème initial.
2. **Programmation dynamique / mémoïsation** : similaire, mais en conservant en mémoire tous les sous-problèmes **non-indépendants** pour éviter de les calculer plusieurs fois.

Sous-problèmes

Définition (Rappel)

Une fonction f est **récursive** si elle s'appelle elle-même.
Elle est composée de :

- Un (ou plusieurs) **cas de base** où f renvoie directement une valeur, sans appel récursif.
- Un (ou plusieurs) **appel récursif** à f avec des arguments plus petits que ceux de l'appel initial, qui garantit de se ramener au cas de base.

Exemple : Calcul de $n! = n \times (n-1)!$

```
def fact(n):
    if n == 0: # cas de base
        return 1
    return n*fact(n - 1) # appel récursif
```

Sous-problèmes

Attention : une fonction récursive peut ne pas terminer (appels récursifs infinis), de même qu'un `while` peut faire boucle infinie.

```
def fact(n):
    return n*fact(n) # oubli du cas de base
```

```
def fact(n):
    if n == 0:
        return 1
    return fact(n) # ne se ramene pas au cas de base
```

Sous-problèmes

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 == 0 or n == 1:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

Complexité : exponentielle...

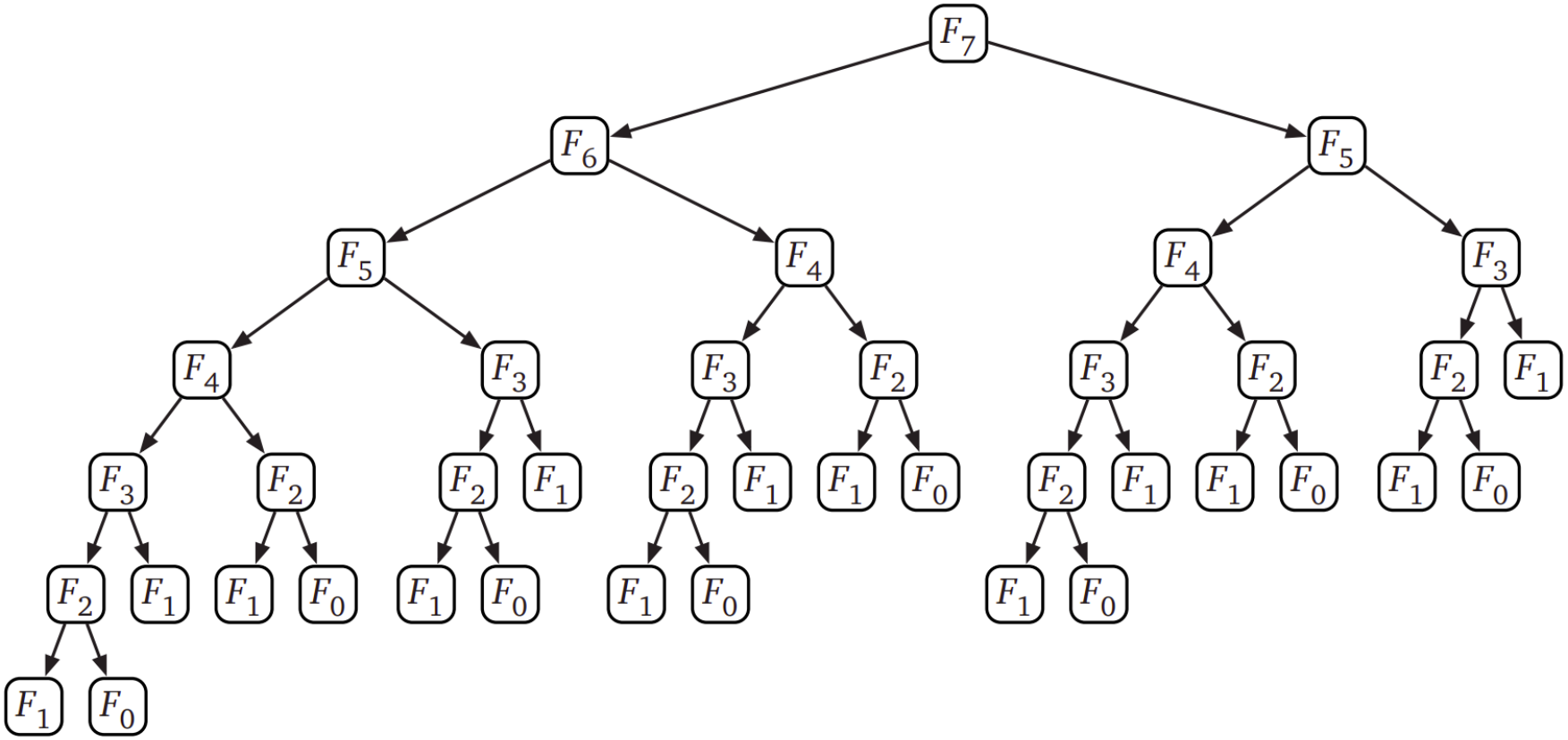
Sous-problèmes

Exercice (pour les 5/2 qui s'ennuient)

Démontrez que la complexité de l'implémentation recursive du calcul des termes de la suite de Fibonacci est en $O(\varphi^n)$.

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):
    L = [1, 1] # L[i] va contenir le ième terme de Fibonacci
    for i in range(n - 1):
        L.append(L[-1] + L[-2]) # récurrence
    return L[n]
```

Sous-problèmes

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

```
def fibo(n):
    L = [1, 1] # L[i] va contenir le ième terme de Fibonacci
    for i in range(n - 1):
        L.append(L[-1] + L[-2]) # récurrence
    return L[n]
```

Complexité : $O(n)$.

Sous-problèmes

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

```
def fibo(n):
    f0, f1 = 1, 1 # f0, f1 sont les deux derniers termes
    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. Déterminer le(s) cas de base(s). Il faut qu'appliquer plusieurs fois l'équation de récurrence ramène à un cas de base.

Programmation dynamique

Pour résoudre un problème de programmation dynamique :

1. Chercher une équation de récurrence. Souvent, cela demande d'introduire un paramètre.
2. Déterminer le(s) cas de base(s). Il faut qu'appliquer plusieurs fois l'équation de récurrence ramène à un cas de base.
3. Stocker en mémoire (dans une liste, matrice ou dictionnaire) les résultats des sous-problèmes pour éviter de les calculer plusieurs fois.

Programmation dynamique

Pour résoudre un problème de programmation dynamique :

1. Chercher une équation de récurrence. Souvent, cela demande d'introduire un paramètre.
2. Déterminer le(s) cas de base(s). Il faut qu'appliquer plusieurs fois l'équation de récurrence ramène à un cas de base.
3. Stocker en mémoire (dans une liste, matrice ou dictionnaire) les résultats des sous-problèmes pour éviter de les calculer plusieurs fois.

Programmation dynamique = récurrence + stocker les résultats intermédiaires.

Programmation dynamique

Exemples :

- Coefficient binomial
- Problème du sac à dos
- Plus courts chemins dans un graphe pondéré : Bellman-Ford et Floyd-Warshall
- Plus longue sous-suite croissante

Programmation dynamique

Exemples :

- Coefficient binomial
- Problème du sac à dos
- Plus courts chemins dans un graphe pondéré : Bellman-Ford et Floyd-Warshall
- Plus longue sous-suite croissante

Remarque : ce sont des exemples suggérés par le programme mais non exigibles.

Coefficient binomial

On veut calculer $\binom{n}{k}$ en utilisant la formule de Pascal :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Exercice

Quel est le ou les cas de base ?

Coefficient binomial

On veut calculer $\binom{n}{k}$ en utilisant la formule de Pascal :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Exercice

Quel est le ou les cas de base ?

En appliquant plusieurs fois l'équation de récurrence, on peut obtenir :

- $k = 0$

Coefficient binomial

On veut calculer $\binom{n}{k}$ en utilisant la formule de Pascal :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Exercice

Quel est le ou les cas de base ?

En appliquant plusieurs fois l'équation de récurrence, on peut obtenir :

- $k = 0$: $\binom{n}{k} = 1$.
- $n = 0$

Coefficient binomial

On veut calculer $\binom{n}{k}$ en utilisant la formule de Pascal :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Exercice

Quel est le ou les cas de base ?

En appliquant plusieurs fois l'équation de récurrence, on peut obtenir :

- $k = 0$: $\binom{n}{k} = 1$.
- $n = 0$: $\binom{n}{k} = 0$ par convention.

Cas particulier : $\binom{0}{0} = 1$ par convention.

Coefficient binomial

Algorithme naïf :

```
def binom(n, k):
    if k == 0: return 1
    if n == 0: return 0
    return binom(n - 1, k - 1) + binom(n - 1, k)
```

Complexité

Coefficient binomial

Algorithme naïf :

```
def binom(n, k):
    if k == 0: return 1
    if n == 0: return 0
    return binom(n - 1, k - 1) + binom(n - 1, k)
```

Complexité : `binom(n, k)` est en $O(\binom{n}{k})$, donc croît très rapidement...

Coefficient binomial

Programmation dynamique :

On utilise une matrice `M` telle que `M[i][j]` contienne $\binom{i}{j}$.

Coefficient binomial

Programmation dynamique :

On utilise une matrice `M` telle que `M[i][j]` contienne $\binom{i}{j}$.

- `M` est de taille $(n+1) \times (k+1)$, car on veut calculer $\binom{i}{j}$ pour $i \in [\![0, n]\!]$ et $j \in [\![0, k]\!]$.

Coefficient binomial

Programmation dynamique :

On utilise une matrice `M` telle que `M[i][j]` contienne $\binom{i}{j}$.

- `M` est de taille $(n+1) \times (k+1)$, car on veut calculer $\binom{i}{j}$ pour $i \in [\![0, n]\!]$ et $j \in [\![0, k]\!]$.
- `M[i][j]` vaut `1` si `j = 0` et `0` si `i = 0`.

Coefficient binomial

Programmation dynamique :

On utilise une matrice `M` telle que `M[i][j]` contienne $\binom{i}{j}$.

- `M` est de taille $(n+1) \times (k+1)$, car on veut calculer $\binom{i}{j}$ pour $i \in [\![0, n]\!]$ et $j \in [\![0, k]\!]$.
- `M[i][j]` vaut `1` si `j = 0` et `0` si `i = 0`.
- On utilise $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ pour remplir `M` de proche en proche.

Coefficient binomial

```python
def binom(n, k):
    M = [[0]*(k + 1) for _ in range(n + 1)]
    for i in range(0, n + 1):
        M[i][0] = 1

    for i in range(1, n + 1):
        for j in range(1, k + 1):
            M[i][j] = M[i - 1][j - 1] + M[i - 1][j]

    return M[n][k]
```

Remarque : On peut aussi définir `M = np.zeros((n + 1, k + 1))`

Complexité : $O(nk)$

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 **mémoïsation** 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 **mémoïsation** 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 un dictionnaire) les arguments pour lesquels la fonction récursive a déjà été calculée.

Mémoïsation

Une **fonction pure** est une fonction vérifiant les deux propriétés suivantes :

Mémoïsation

Une **fonction pure** est une fonction vérifiant les deux propriétés suivantes :

- Sa valeur de retour est invariante pour les mêmes arguments,

Mémoïsation

Une **fonction pure** est une fonction vérifiant les deux propriétés suivantes :

- Sa valeur de retour est invariante pour les mêmes arguments,
- Son évaluation ne provoque pas d'effets de bord.

Mémoïsation

Une **fonction pure** est une fonction vérifiant les deux propriétés suivantes :

- Sa valeur de retour est invariante pour les mêmes arguments,
- Son évaluation ne provoque pas d'effets de bord.

C'est une fonction au sens qu'on lui prête en mathématiques.

Seule une fonction pure peut être mémoïsée.

Mémoïsation

Version mémoïsée du calcul de la suite de Fibonacci :

```
def fibo(n):
    d = {} # d[k] contiendra le kème terme de la suite
    def aux(k):
        if k == 0 or k == 1:
            return 1
        if k not in d:
            d[k] = aux(k - 1) + aux(k - 2)
        return d[k]
    return aux(n)
```

Mémoïsation

Mémoïsation du calcul de coefficient binomial :

```
def binom(n, k):
    d = {}
    def aux(i, j):
        if j == 0: return 1
        if i == 0: return 0
        if (i, j) not in d:
            d[(i, j)] = aux(i - 1, j - 1) + aux(i - 1, j)
        return d[(i, j)]
    return aux(n, k)
```

Mémoïsation

Il est possible de « mémoïser » automatiquement une fonction.
En Python 3.10 :

```python
from functools import cache

@cache
def f(n):
    if n <= 1:
        return n
    return f(n-1) + f(n-2)
```

Mémoïsation

Il est possible de « mémoïser » automatiquement une fonction.
En Python 3.10 :

```python
from functools import cache

@cache
def f(n):
    if n <= 1:
        return n
    return f(n-1) + f(n-2)
```

Mémoïsation du calcul de coefficient binomial :

```python
@cache
def binom(n, k):
    if k == 0: return 1
    if n == 0: return 0
    return binom(n - 1, k - 1) + binom(n - 1, k)
```

Problème du sac à dos

Problème du sac à dos

Entrée : une capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n.
Sortie : la valeur maximum que l'on peut mettre dans un sac de capacité c.

Problème du sac à dos

Problème du sac à dos

Entrée : une capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n.
Sortie : la valeur maximum que l'on peut mettre dans un sac de capacité c.

Exemple

On considère un sac à dos de capacité 10kg et les objets suivants :

poids (kg)	2	2	2	3	5	5	8
valeur (€)	1	1	1	7	10	10	13

Quelle est la valeur maximum que l'on peut mettre dans le sac ?

Problème du sac à dos

Définition

Un algorithme est dit **glouton** s'il prend toujours la meilleure décision à chaque étape (localement), sans se soucier des conséquences futures.

Problème du sac à dos

Définition

Un algorithme est dit **glouton** s'il prend toujours la meilleure décision à chaque étape (localement), sans se soucier des conséquences futures.

On peut imaginer différents algorithmes gloutons, selon la façon dont on va sélectionner le prochain objet à mettre dans le sac :

1. Poids minimum.

Problème du sac à dos

Définition

Un algorithme est dit **glouton** s'il prend toujours la meilleure décision à chaque étape (localement), sans se soucier des conséquences futures.

On peut imaginer différents algorithmes gloutons, selon la façon dont on va sélectionner le prochain objet à mettre dans le sac :

1. Poids minimum.
2. Valeur maximum.

Problème du sac à dos

Définition

Un algorithme est dit **glouton** s'il prend toujours la meilleure décision à chaque étape (localement), sans se soucier des conséquences futures.

On peut imaginer différents algorithmes gloutons, selon la façon dont on va sélectionner le prochain objet à mettre dans le sac :

1. Poids minimum.
2. Valeur maximum.
3. Ratio valeur/poids maximum.

Malheureusement, ces algorithmes gloutons ne donnent pas toujours l'optimum.

Problème du sac à dos

Problème du sac à dos

Entrée : un sac à dos de capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n. On suppose que les poids sont strictement positifs.
Sortie : la valeur maximum que l'on peut mettre dans le sac.

Problème du sac à dos

Problème du sac à dos

Entrée : un sac à dos de capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n. On suppose que les poids sont strictement positifs.
Sortie : la valeur maximum que l'on peut mettre dans le sac.

Soit $dp[i][j]$ la valeur maximum que l'on peut mettre dans un sac de capacité i, en ne considérant que les objets o_1, ..., o_j.

Problème du sac à dos

Problème du sac à dos

Entrée : un sac à dos de capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n. On suppose que les poids sont strictement positifs.
Sortie : la valeur maximum que l'on peut mettre dans le sac.

Soit $dp[i][j]$ la valeur maximum que l'on peut mettre dans un sac de capacité i, en ne considérant que les objets o_1, ..., o_j.

$$dp[i][0] = 0$$

$$dp[i][j] = \max\Big(\underbrace{dp[i][j-1]}_{\text{sans prendre } o_j}, \underbrace{dp[i-w_j][j-1] + v_j}_{\text{en prenant } o_j, \text{si } i - w_j \geq 0} \Big)$$

Problème du sac à dos

Résolution du sac à dos par programmation dynamique

Pour $i = 0$ à c:
    $dp[i][0] \leftarrow 0$

Pour $i = 1$ à c:
    Pour $j = 1$ à n:
        Si $i - w_j \geq 0$:
            $dp[i][j] \leftarrow \max\left(dp[i][j-1], dp[i-w_j][i-1] + v_j\right)$
        Sinon:
            $dp[i][j] \leftarrow dp[i][j-1]$

Complexité :

Problème du sac à dos

Résolution du sac à dos par programmation dynamique

Pour $i = 0$ à c:
    $dp[i][0] \leftarrow 0$

Pour $i = 1$ à c:
    Pour $j = 1$ à n:
        Si $i - w_j \geq 0$:
            $dp[i][j] \leftarrow \max\left(dp[i][j-1], dp[i-w_j][i-1] + v_j\right)$
        Sinon:
            $dp[i][j] \leftarrow dp[i][j-1]$

Complexité : $O(nc)$

Problème du sac à dos

```
def knapsack(c, w, v):
    """
    Renvoie la valeur maximum que l'on peut mettre
    dans un sac à dos de capacité c.
    Le ième objet a pour poids w[i] et valeur v[i].
    """
    n = len(w) # nombre d'objets
    dp = [[0]*(n + 1) for i in range(c + 1)]
    # dp[i][j] = valeur max dans un sac de capacité i
    # où j est le nombre d'objets autorisés
    for i in range(1, c + 1):
        for j in range(1, n + 1):
            if w[j - 1] <= i:
                x = v[j - 1] + dp[i - w[j - 1]][j - 1]
                dp[i][j] = max(dp[i][j - 1], x)
            else:
                dp[i][j] = dp[i][j - 1]
    return dp[c][n]
```

Problème du sac à dos

Comme on a juste besoin de stocker `dp[...][j - 1]` pour calculer `dp[...][j]`, on peut simplifier :

Problème du sac à dos

Comme on a juste besoin de stocker `dp[...][j - 1]` pour calculer `dp[...][j]`, on peut simplifier :

```
def knapsack2(c, w, v):
    n = len(w)
    dp = [0]*(c + 1)
    for j in range(n):
        dp_ = dp[:] # copie de dp
        for i in range(c + 1):
            if w[j] <= i:
                dp[i] = max(dp_[i], v[j] + dp_[i - w[j]])
    return dp[-1]
```

Problème du sac à dos

Version récursive et mémoïsée pour le sac à dos :

```
def knapsack_memo(c, w, v):
    dp = {}
    def aux(i, j):
        if i == 0 or j == 0:
            return 0
        if (i, j) not in dp:
            dp[(i, j)] = aux(i, j - 1)
            if w[j - 1] <= i:
                x = v[j - 1] + aux(i - w[j - 1], j - 1)
                dp[(i, j)] = max(dp[(i, j)], x)
        return dp[(i, j)]
    return aux(c, len(w))
```