Programmation dynamique & Algorithmes gloutons

CPES2: Algorithmique et Applications

Pierre Senellart

27 septembre 2021

Plan

Rendu de monnaie

- On fixe $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ la valeur de n types de pièces de monnaie
- Par exemple, dans la zone euro :

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	2	5	10	20	50	100	200

- Combien y a-t-il de manières de faire l'appoint pour une somme $S \in \mathbb{N}$?

Rendu de monnaie

- On fixe $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ la valeur de n types de pièces de monnaie
- Par exemple, dans la zone euro :

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	2	5	10	20	50	100	200

- Combien y a-t-il de manières de faire l'appoint pour une somme $S \in \mathbb{N}$?
- Par exemple, si $S = 7$, 6 manières différentes :

Rendu de monnaie

- On fixe $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ la valeur de n types de pièces de monnaie
- Par exemple, dans la zone euro :

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	2	5	10	20	50	100	200

- Combien y a-t-il de manières de faire l'appoint pour une somme $S \in \mathbb{N}$?
- Par exemple, si $S = 7$, 6 manières différentes :

$$\begin{cases} 1+1+1+1+1+1+1 \\ \\ \\ \\ \\ \\ \end{cases}$$

Rendu de monnaie

- On fixe $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ la valeur de n types de pièces de monnaie
- Par exemple, dans la zone euro :

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	2	5	10	20	50	100	200

- Combien y a-t-il de manières de faire l'appoint pour une somme $S \in \mathbb{N}$?
- Par exemple, si $S = 7$, 6 manières différentes :

$$\begin{cases} 1+1+1+1+1+1+1 \\ 1+1+1+1+1+2 \end{cases}$$

Rendu de monnaie

- On fixe $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ la valeur de n types de pièces de monnaie
- Par exemple, dans la zone euro :

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	2	5	10	20	50	100	200

- Combien y a-t-il de manières de faire l'appoint pour une somme $S \in \mathbb{N}$?
- Par exemple, si $S = 7$, 6 manières différentes :

$$\begin{cases} 1+1+1+1+1+1+1 \\ 1+1+1+1+1+2 \\ 1+1+1+2+2 \end{cases}$$

Rendu de monnaie

- On fixe $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ la valeur de n types de pièces de monnaie
- Par exemple, dans la zone euro :

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	2	5	10	20	50	100	200

- Combien y a-t-il de manières de faire l'appoint pour une somme $S \in \mathbb{N}$?
- Par exemple, si $S = 7$, 6 manières différentes :

$$\begin{cases} 1+1+1+1+1+1+1 \\ 1+1+1+1+1+2 \\ 1+1+1+2+2 \\ 1+1+5 \end{cases}$$

Rendu de monnaie

- On fixe $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ la valeur de n types de pièces de monnaie
- Par exemple, dans la zone euro :

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	2	5	10	20	50	100	200

- Combien y a-t-il de manières de faire l'appoint pour une somme $S \in \mathbb{N}$?
- Par exemple, si $S = 7$, 6 manières différentes :

$$\begin{cases} 1+1+1+1+1+1+1 \\ 1+1+1+1+1+2 \\ 1+1+1+2+2 \\ 1+1+5 \\ 1+2+2+2 \end{cases}$$

Rendu de monnaie

- On fixe $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ la valeur de n types de pièces de monnaie
- Par exemple, dans la zone euro :

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	2	5	10	20	50	100	200

- Combien y a-t-il de manières de faire l'appoint pour une somme $S \in \mathbb{N}$?
- Par exemple, si $S = 7$, 6 manières différentes :

$$\begin{cases} 1+1+1+1+1+1+1 \\ 1+1+1+1+1+2 \\ 1+1+1+2+2 \\ 1+1+5 \\ 1+2+2+2 \\ 2+5 \end{cases}$$

Solution par force brute

- Essayer toutes les combinaisons possibles, vérifier lesquelles somment à S

Solution par force brute

- Essayer toutes les combinaisons possibles, vérifier lesquelles somment à S
- Par exemple, pour $S = 7$, on peut essayer les combinaisons avec $\leqslant 7$ pièces de 1, $\leqslant 3$ pièces de 2, $\leqslant 1$ pièce de 5

Solution par force brute

- Essayer toutes les combinaisons possibles, vérifier lesquelles somment à S
- Par exemple, pour $S = 7$, on peut essayer les combinaisons avec $\leqslant 7$ pièces de 1, $\leqslant 3$ pièces de 2, $\leqslant 1$ pièce de 5
- 8 possibilités pour les pièces de 1, 4 possibilités pour les pièces de 2, 2 possibilités pour les pièces de 5, donc $8 \times 4 \times 2 = 64$ possibilités à tester

Solution par force brute

- Essayer toutes les combinaisons possibles, vérifier lesquelles somment à S
- Par exemple, pour $S = 7$, on peut essayer les combinaisons avec $\leqslant 7$ pièces de 1, $\leqslant 3$ pièces de 2, $\leqslant 1$ pièce de 5
- 8 possibilités pour les pièces de 1, 4 possibilités pour les pièces de 2, 2 possibilités pour les pièces de 5, donc $8 \times 4 \times 2 = 64$ possibilités à tester
- Dans le cas général, algorithme en $\Theta\left(\mathrm{E}\left[\frac{S}{p_1}\right] \times \cdots \times \mathrm{E}\left[\frac{S}{p_n}\right]\right) = O(S^n)$

Solution par force brute

- Essayer toutes les combinaisons possibles, vérifier lesquelles somment à S
- Par exemple, pour $S = 7$, on peut essayer les combinaisons avec $\leqslant 7$ pièces de 1, $\leqslant 3$ pièces de 2, $\leqslant 1$ pièce de 5
- 8 possibilités pour les pièces de 1, 4 possibilités pour les pièces de 2, 2 possibilités pour les pièces de 5, donc $8 \times 4 \times 2 = 64$ possibilités à tester
- Dans le cas général, algorithme en $\Theta\left(\mathrm{E}\left[\frac{S}{p_1}\right] \times \cdots \times \mathrm{E}\left[\frac{S}{p_n}\right]\right) = O(S^n)$
- Peut-on mieux faire ?

Formule de récurrence

- Pour $S \in \mathbb{Z}$ et $k \in \mathbb{N}$, on note $N(S, k)$ le nombre de combinaisons pour le rendu de monnaie sur la somme S avec les k premières pièces

Formule de récurrence

- Pour $S \in \mathbb{Z}$ et $k \in \mathbb{N}$, on note $N(S, k)$ le nombre de combinaisons pour le rendu de monnaie sur la somme S avec les k premières pièces
- On a des cas très simples (de base) :

$$\begin{cases} N(0, k) = 1 & \text{pour tout } k \\ N(S, 0) = 0 & \text{pour tout } S > 0 \\ N(S, k) = 0 & \text{pour tout } S < 0 \text{ et pour tout } k \end{cases}$$

Formule de récurrence

- Pour $S \in \mathbb{Z}$ et $k \in \mathbb{N}$, on note $N(S,k)$ le nombre de combinaisons pour le rendu de monnaie sur la somme S avec les k premières pièces
- On a des cas très simples (de base) :

$$\begin{cases} N(0,k) = 1 & \text{pour tout } k \\ N(S,0) = 0 & \text{pour tout } S > 0 \\ N(S,k) = 0 & \text{pour tout } S < 0 \text{ et pour tout } k \end{cases}$$

- On peut résoudre un cas complexe à l'aide de cas plus simples :

$$N(S,k) = N(S,k-1) + N(S-p_k,k) \quad \text{pour tous } S > 0,\ k > 0$$

Application au cas $S = 7$

	k			
S	0	1	2	3
0				
1				
2				
3				
4				
5				
6				
7				

p_1	p_2	p_3
1	2	5

$$\begin{cases} N(0,k) = 1 \\ N(S,0) = 0 \\ N(S,k) = 0 \text{ pour } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) \end{cases}$$

Application au cas $S = 7$

	k			
S	0	1	2	3
0	1	1	1	1
1	0			
2	0			
3	0			
4	0			
5	0			
6	0			
7	0			

p_1	p_2	p_3
1	2	5

$$\begin{cases} N(0,k) = 1 \\ N(S,0) = 0 \\ N(S,k) = 0 \text{ pour } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) \end{cases}$$

Application au cas $S = 7$

	k			
S	0	1	2	3
0	1	1	1	1
1	0	1		
2	0			
3	0			
4	0			
5	0			
6	0			
7	0			

p_1	p_2	p_3
1	2	5

$$\begin{cases} N(0,k) = 1 \\ N(S,0) = 0 \\ N(S,k) = 0 \text{ pour } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) \end{cases}$$

Application au cas $S = 7$

	k			
S	0	1	2	3
0	1	1	1	1
1	0	1	1	
2	0			
3	0			
4	0			
5	0			
6	0			
7	0			

p_1	p_2	p_3
1	2	5

$$\begin{cases} N(0,k) = 1 \\ N(S,0) = 0 \\ N(S,k) = 0 \text{ pour } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) \end{cases}$$

Application au cas $S = 7$

	k			
S	0	1	2	3
0	1	1	1	1
1	0	1	1	1
2	0			
3	0			
4	0			
5	0			
6	0			
7	0			

p_1	p_2	p_3
1	2	5

$$\begin{cases} N(0,k) = 1 \\ N(S,0) = 0 \\ N(S,k) = 0 \text{ pour } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) \end{cases}$$

Application au cas $S = 7$

	k			
S	0	1	2	3
0	1	1	1	1
1	0	1	1	1
2	0	1	2	2
3	0			
4	0			
5	0			
6	0			
7	0			

p_1	p_2	p_3
1	2	5

$$\begin{cases} N(0,k) = 1 \\ N(S,0) = 0 \\ N(S,k) = 0 \text{ pour } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) \end{cases}$$

Application au cas $S = 7$

	k			
S	0	1	2	3
0	1	1	1	1
1	0	1	1	1
2	0	1	2	2
3	0	1	2	2
4	0			
5	0			
6	0			
7	0			

p_1	p_2	p_3
1	2	5

$$\begin{cases} N(0,k) = 1 \\ N(S,0) = 0 \\ N(S,k) = 0 \text{ pour } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) \end{cases}$$

Application au cas $S = 7$

	k			
S	0	1	2	3
0	1	1	1	1
1	0	1	1	1
2	0	1	2	2
3	0	1	2	2
4	0	1	3	3
5	0			
6	0			
7	0			

p_1	p_2	p_3
1	2	5

$$\begin{cases} N(0,k) = 1 \\ N(S,0) = 0 \\ N(S,k) = 0 \text{ pour } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) \end{cases}$$

Application au cas $S = 7$

	k			
S	0	1	2	3
0	1	1	1	1
1	0	1	1	1
2	0	1	2	2
3	0	1	2	2
4	0	1	3	3
5	0	1	3	4
6	0			
7	0			

p_1	p_2	p_3
1	2	5

$$\begin{cases} N(0,k) = 1 \\ N(S,0) = 0 \\ N(S,k) = 0 \text{ pour } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) \end{cases}$$

Application au cas $S = 7$

	k			
S	0	1	2	3
0	1	1	1	1
1	0	1	1	1
2	0	1	2	2
3	0	1	2	2
4	0	1	3	3
5	0	1	3	4
6	0	1	4	5
7	0			

p_1	p_2	p_3
1	2	5

$$\begin{cases} N(0,k) = 1 \\ N(S,0) = 0 \\ N(S,k) = 0 \text{ pour } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) \end{cases}$$

Application au cas $S = 7$

	k			
S	0	1	2	3
0	1	1	1	1
1	0	1	1	1
2	0	1	2	2
3	0	1	2	2
4	0	1	3	3
5	0	1	3	4
6	0	1	4	5
7	0	1	4	6

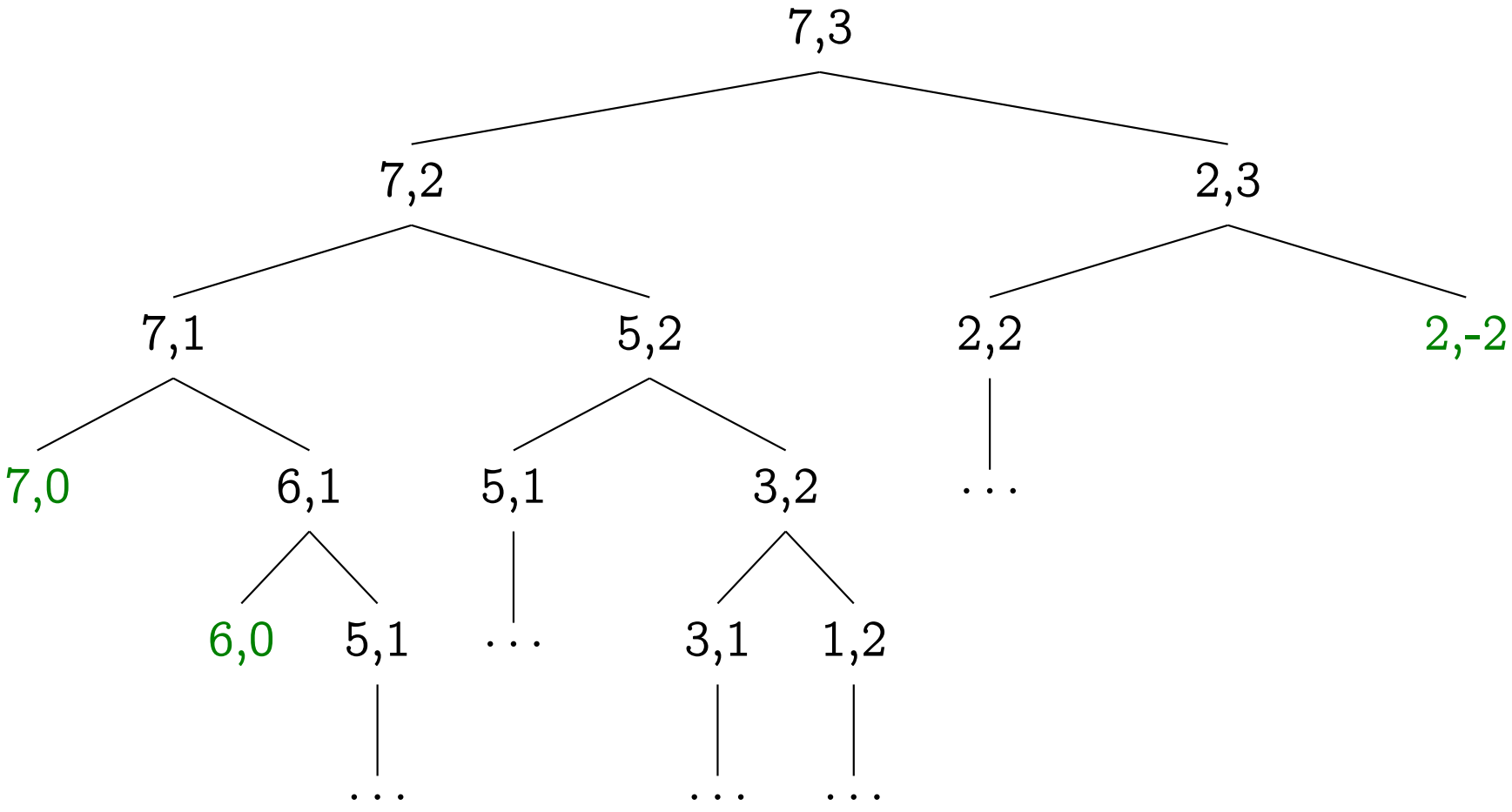
p_1	p_2	p_3
1	2	5

$$\begin{cases} N(0,k) = 1 \\ N(S,0) = 0 \\ N(S,k) = 0 \text{ pour } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) \end{cases}$$

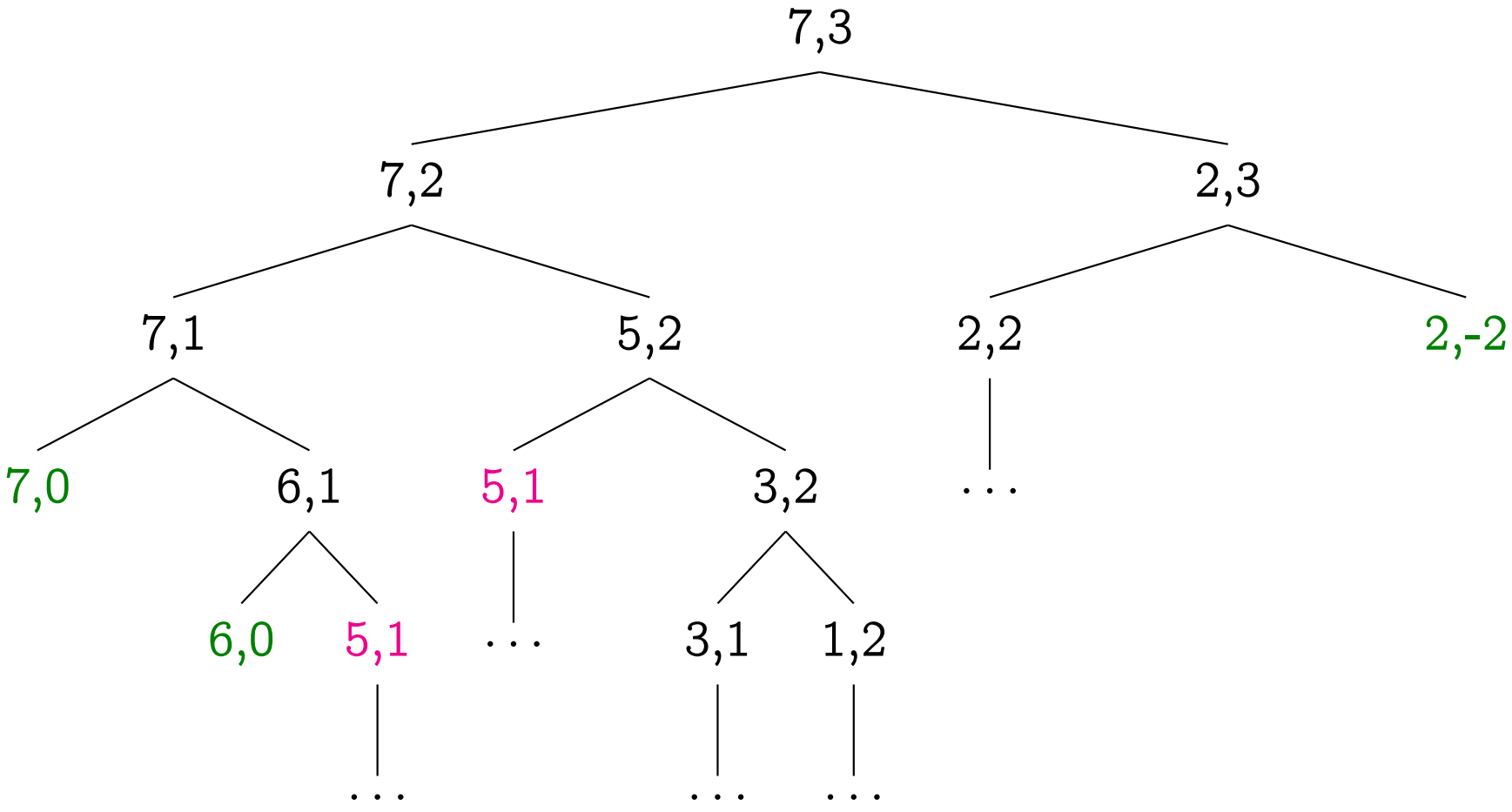
Solution par fonction récursive

```
function nb_pieces(S, k)
if S = 0 then
    return 1;
else
    if S < 0 or k = 0 then
        return 0;
    else
        return nb_pieces(S, k − 1) + nb_pieces(S − p_k, k);
```

Appels récursifs pour $S = 7$, $p = \{1, 2, 3\}$



Appels récursifs pour $S = 7$, $p = \{1, 2, 3\}$



Fonction récursive : complexité

- Pour simplifier l'étude, on regarde le cas (défavorable mais possible) où $p = \{1, \ldots, 1\}$

Fonction récursive : complexité

- Pour simplifier l'étude, on regarde le cas (défavorable mais possible) où $p = \{1, \ldots, 1\}$
- En posant $N = S + n$, on a :

$$T(N) = \begin{cases} O(1) & \text{si } S \leqslant 0 \text{ ou } n = 0 \\ T(N-1) + T(N-1) + O(1) & \text{sinon} \end{cases}$$

Fonction récursive : complexité

- Pour simplifier l'étude, on regarde le cas (défavorable mais possible) où $p = \{1, \ldots, 1\}$
- En posant $N = S + n$, on a :

$$T(N) = \begin{cases} O(1) & \text{si } S \leqslant 0 \text{ ou } n = 0 \\ T(N-1) + T(N-1) + O(1) & \text{sinon} \end{cases}$$

- On verra plus tard comment résoudre ce genre de récurrences, mais on observe que $T(N) \approx 2 \times T(N-1)$, d'où $T(N) = \Theta\left(2^N\right) = \Theta\left(2^{S+n}\right)$.

Fonction récursive : complexité

- Pour simplifier l'étude, on regarde le cas (défavorable mais possible) où $p = \{1, \ldots, 1\}$
- En posant $N = S + n$, on a :

$$T(N) = \begin{cases} O(1) & \text{si } S \leqslant 0 \text{ ou } n = 0 \\ T(N-1) + T(N-1) + O(1) & \text{sinon} \end{cases}$$

- On verra plus tard comment résoudre ce genre de récurrences, mais on observe que $T(N) \approx 2 \times T(N-1)$, d'où $T(N) = \Theta\left(2^N\right) = \Theta\left(2^{S+n}\right)$.
- C'est pire que la force brute qui était en $O(S^n)$!

Solution avec mémoïsation

function nb_pieces(S, k)
if $M(S,k)$ *est défini* then
  return $M(S,k)$;
if $S = 0$ then
  $M(S,k) \leftarrow 1$;
else
  if $S < 0$ or $k = 0$ then
    $M(S,k) \leftarrow 0$;
  else
    $M(S,k) \leftarrow \text{nb\_pieces}(S, k-1) + \text{nb\_pieces}(S - p_k, k)$;
return $M(S,k)$;

Mémoïsation : complexité

- Chaque appel est en $O(1)$, sauf au plus une fois par case du tableau M, de taille $(S + p_n) \times (n + 1)$
- On a donc une complexité en $O(S \times n)$: nettement plus efficace !

Solution par programmation dynamique

```
function nb_pieces(S, n)
T ← creer_tableau(S + 1, n + 1);
for i ← 1 to S do
    T(i, 0) ← 0;
for i ← 0 to n do
    T(0, i) ← 1;
for i ← 1 to n do
    for j ← 1 to S do
        if p_j ⩽ i then
            T(i, j) ←
              T(i, j − 1) + T(i − p_j, j);
        else
            T(i, j) ← T(i, j − 1);
return T(S, n);
```

Solution par programmation dynamique

```
function nb_pieces(S, n)
T ← creer_tableau(S + 1, n + 1);
for i ← 1 to S do
    T(i, 0) ← 0;
for i ← 0 to n do
    T(0, i) ← 1;
for i ← 1 to n do
    for j ← 1 to S do
        if p_j ⩽ i then
            T(i, j) ←
              T(i, j − 1) + T(i − p_j, j);
        else
            T(i, j) ← T(i, j − 1);
return T(S, n);
```

Complexité :

$\Theta(S \times n)$

Variante : Nombre minimum de pièces

- Et si on veut le nombre minimum de pièces à rendre, plutôt que le nombre de combinaison de pièces ?

Variante : Nombre minimum de pièces

- Et si on veut le nombre minimum de pièces à rendre, plutôt que le nombre de combinaison de pièces ?
- On établit une nouvelle formule de récurrence :

$$\begin{cases} N(0,k) = 0 & \text{pour tout } k \\ N(S,0) = +\infty & \text{pour tout } S \neq 0 \\ N(S,k) = \min(N(S,k-1), N(S-p_k,k)+1) & \text{pour tous } S,k > 0 \end{cases}$$

Variante : Nombre minimum de pièces

- Et si on veut le nombre minimum de pièces à rendre, plutôt que le nombre de combinaison de pièces ?
- On établit une nouvelle formule de récurrence :

$$\begin{cases} N(0,k) = 0 & \text{pour tout } k \\ N(S,0) = +\infty & \text{pour tout } S \neq 0 \\ N(S,k) = \min(N(S,k-1), N(S-p_k,k)+1) & \text{pour tous } S,k > 0 \end{cases}$$

- On peut utiliser cette formule de récurrence pour écrire des solutions récursive, avec mémoïsation, par programmation dynamique...

Plan

Cas d'application de la programmation dynamique

- Quand peut-on et est-il intéressant d'utiliser de la programmation dynamique (ou de la mémoïsation) ?

Cas d'application de la programmation dynamique

- Quand peut-on et est-il intéressant d'utiliser de la programmation dynamique (ou de la mémoïsation) ?
- On a besoin de deux conditions :

Cas d'application de la programmation dynamique

- Quand peut-on et est-il intéressant d'utiliser de la programmation dynamique (ou de la mémoïsation) ?
- On a besoin de deux conditions :

  Sous-structures optimales. Les solutions optimales des sous-problèmes sont des sous-structures de la solution optimale du problème orginal.

Cas d'application de la programmation dynamique

- Quand peut-on et est-il intéressant d'utiliser de la programmation dynamique (ou de la mémoïsation) ?
- On a besoin de deux conditions :

  Sous-structures optimales. Les solutions optimales des sous-problèmes sont des sous-structures de la solution optimale du problème orginal.

  Sous-problèmes répétés. On rencontre de nombreuses fois le même problème.

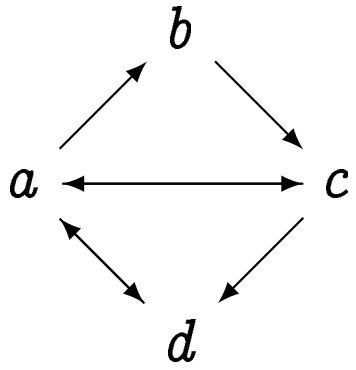
Cas d'application de la programmation dynamique

- Quand peut-on et est-il intéressant d'utiliser de la programmation dynamique (ou de la mémoïsation) ?
- On a besoin de deux conditions :

  Sous-structures optimales. Les solutions optimales des sous-problèmes sont des sous-structures de la solution optimale du problème orginal.

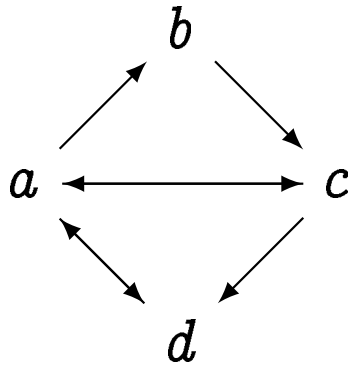
  Sous-problèmes répétés. On rencontre de nombreuses fois le même problème.
- Si on n'a que les sous-structures optimales, mais pas les sous-problèmes répétés, la récursion simple suffit (voir le cours sur *Diviser pour régner*)

Exemple d'application : Chemins dans les graphes



- Plus court chemin :
  - Chemin d'un nœud à un autre avec le plus petit nombre d'arêtes
  - Sous-structures optimales : Le plus court chemin de x à y en passant par z est formé du plus court chemin de x à z combiné au plus court chemin de z à y
  - Sous-problèmes répétés : Le plus court chemin de x à y est utile au calcul de nombreux plus courts chemins
  - La programmation dynamique est appropriée !

Exemple d'application : Chemins dans les graphes



- Plus long chemin (sans cycle) :
  - Chemin d'un nœud à un autre avec le plus grand nombre d'arêtes, sans passer deux fois par le même nœud
  - Sous-structures non-optimales : Le plus long chemin de x à y en passant par z n'est pas nécessairement formé du plus long chemin de x à z combiné au plus long chemin de z à y
  - Exemple : b à d en passant par a
  - La programmation dynamique ne marche pas !

Exemple d'application : Sac à dos 0–1 (1/3)

- On se donne n objets $o_1, \ldots, o_n$, chaque objet o_i ayant un poids w_i et une valeur v_i
- On dispose d'un sac à dos de capacité maximale (en poids) W
- Comment maximiser la valeur des objets du sac à dos en conservant un poids total $\leqslant W$?
- On cherche $S \subseteq \{1, \ldots, n\}$ tel que :

$$\sum_{i \in S} w_i \leqslant W \text{ et } \sum_{i \in S} v_i \text{ maximale}$$

Exemple d'application : Sac à dos 0–1 (2/3)

	o_1	o_2	o_3
poids	10	20	30
valeur	60	110	150
ratio	6	5.5	5

$W = 50$

- Méthode gloutonne (*greedy*) : prendre les objets par ordre décroissant de ratio valeur/poids, tant que la capacité le permet.
- Ne fonctionne pas ! Conduit à prendre o_1 et o_2, avec une valeur totale de 170 vs 260 pour o_2 et o_3
- Le meilleur choix local (o_1) n'est pas le meilleur choix global

Exemple d'application : Sac à dos 0–1 (3/3)

- La programmation dynamique fonctionne (sous-structures optimales, sous-problèmes répétés)

Exemple d'application : Sac à dos 0–1 (3/3)

- La programmation dynamique fonctionne (sous-structures optimales, sous-problèmes répétés)
- On pose $V(W, k)$ la valeur maximale atteignable avec une capacité de W et les k premiers objets

Exemple d'application : Sac à dos 0–1 (3/3)

- La programmation dynamique fonctionne (sous-structures optimales, sous-problèmes répétés)
- On pose $V(W, k)$ la valeur maximale atteignable avec une capacité de W et les k premiers objets
- Alors :

$$V(W,k) = \begin{cases} 0 & \text{si } k = 0 \\ \max(V(W - w_k, k-1) + v_k, & \\ \quad V(W, k-1)) & \text{si } w_k \leqslant W \\ V(W, k-1) & \text{sinon} \end{cases}$$

Exemple d'application : Sac à dos 0–1 (3/3)

- La programmation dynamique fonctionne (sous-structures optimales, sous-problèmes répétés)
- On pose $V(W, k)$ la valeur maximale atteignable avec une capacité de W et les k premiers objets
- Alors :

$$V(W,k) = \begin{cases} 0 & \text{si } k = 0 \\ \max(V(W - w_k, k-1) + v_k, & \\ \quad V(W, k-1)) & \text{si } w_k \leqslant W \\ V(W, k-1) & \text{sinon} \end{cases}$$

- En programmation dynamique, complexité de $\Theta(W \times n)$

Plan

Plan

Quand peut-on utiliser la récursion ?

Un problème P peut être résolu par récursion quand :

- On peut le paramétrer par un ou plusieurs entiers $P(n_1 \dots n_k)$
- La solution de $P(n_1 \dots n_k)$ peut être obtenue à partir de la solution de $P(n_1^{(1)} \dots n_k^{(1)}) \dots P(n_1^{(\ell)} \dots n_k^{(\ell)})$ pour un certain $\ell \geqslant 1$ avec, pour tout $1 \leqslant i \leqslant \ell$, $n_j^{(i)} \leqslant n_j$ où $1 \leqslant j \leqslant k$, l'une de ces inégalités étant strictes $(\forall i, \exists j, n_j^{(i)} < n_j)$: cas récursif
- $P(0 \dots 0)$ (ou $P(n_1 \dots n_k)$ pour n_i « petit » en fonction du cas) facile à calculer : cas de base

Exemple 1 : Factorielle

- $P(n) = n!$
- $k = 1, \ell = 1$
- $P(n) = n \times P(n-1)$ pour $n \geqslant 1$
- $P(0) = 1$

Exemple 2 : Fibonacci

- $P(n)$ n^{e} nombre de Fibonacci
- $k = 1$, $\ell = 2$
- $P(n) = P(n-1) + P(n-2)$ for $n \geqslant 1$
- $P(0) = 1$, $P(1) = 1$

Exemple 3 : Distance d'édition de Levenshtein

$d(s, s')$ distance d'édition (nombre minimal de caractères à ajouter, enlever, modifier) pour aller d'une chaîne s à une chaîne s'.

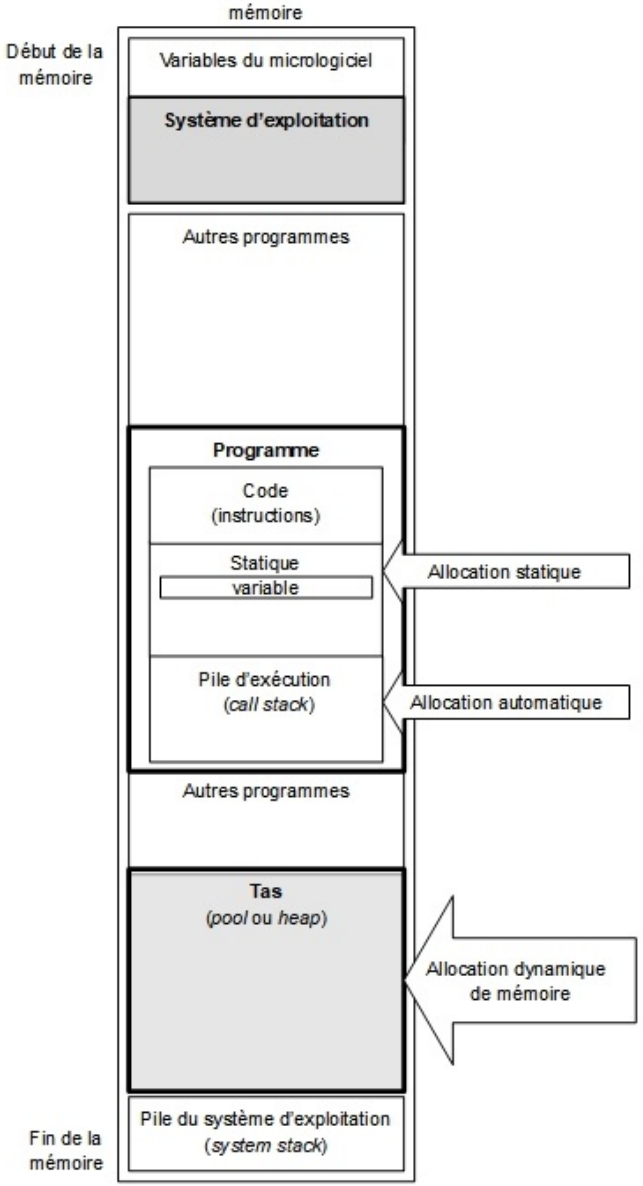
- $P(n_1, n_2)$ distance d'édition entre le préfixe de longueur n_1 de s et le préfixe de longueur n_2 de s'
- $k = 2$, $\ell = 3$
- $P(n_1, n_2) =$
  $\min\left(P(n_1 - 1, n_2) + 1, P(n_1, n_2 - 1) + 1, P(n_1 - 1, n_2 - 1) + \mathbb{I}_{s_{n_1} \neq s'_{n_2}}\right)$
  for $n_1 \geqslant 1$, $n_2 \geqslant 1$
- $P(n_1, 0) = n_1$ pour $n_1 \geqslant 0$; $P(0, n_2) = n_2$ pour $n_2 \geqslant 0$.

$\mathbb{I}_b$ est 1 si b est vrai, 0 sinon

Pseudo-code

- Fonction récursive
  function $P(n_1, \ldots, n_k)$
  if *cas de base* then
  | ...;
  | return ...;
  else
  | // cas récursif
  | ...;
  | // ℓ appels à P
  | return ...;
- La récursion est terminale si $\ell = 1$ et l'appel à P est la dernière instruction du cas récursif (return $P(n'_1, \ldots, n'_k)$)
- Remarque : sur l'exemple plus haut, le else peut être omis

Aparté : pile et tas



Complexité

- Borne supérieure (parfois mieux, p. ex., si le cas récursif est sur $\frac{n}{2}$ et non sur $n-1$) :
  - Si $\ell = 1$: $O(n_1 + \cdots + n_k)$: souvent acceptable
  - Si $\ell > 1$: $O(\ell^{n_1+\cdots+n_k})$: souvent déraisonnable
- Attention à la mémoire utilisée sur la pile, $O(\sum_{i=1}^{k} n_i)$ (souvent, seulement quelques centaines de kilo-octets ou quelques méga-octets disponibles sur la pile) et des limites du langage (en Python, `sys.getrecursionlimit()` donne la limite du nombre de récursion imbriquée, typiquement 1000)
- Quand la récursion est terminale, le compilateur peut réécrire la récursion en itération, ce qui enlève les problèmes de gestion de pile ; mais pas en Python ! (décision de Guido van Rossum)

Plan

Idée (1/3)

Cache : objet global (ou propriété statique de classe) qui se souvient des résultats de la fonction P d'un appel à l'autre

Idée (2/3)

function $P(n_1, \ldots, n_k)$
if $M(n_1, \ldots, n_k)$ *est défini* then
  return $M(n_1, \ldots, n_k)$;
if *cas de base* then
  $\ldots$;
  $r \leftarrow \ldots$;
else
  // cas récursif
  $\ldots$;
  // ℓ appels à P
  $r \leftarrow \ldots$;
$M(n_1, \ldots, n_k) \leftarrow r$;
return r;

Idée (3/3)

- Plus possible d'avoir une récursion terminale !
- Structure de données : en Python, `array.array`, liste Python, ou dictionnaire suivant les éléments à stocker
- Utiliser un `array.array` ou liste Python quand les paramètres sont des entiers à valeurs contiguës, un dictionnaire sinon

Complexité

- $O(n_1 \times \cdots \times n_k \times \ell)$ opérations
- $O(n_1 \times \cdots \times n_k)$ mémoire sur le tas (espace précis dépendant de la structure de données)
- $O(n_1 + \cdots + n_k)$ mémoire sur la pile
- Pas de calcul inutile !

Plan

Idée

- On construit un tableau multi-dimensionnel $n_1 \times \cdots \times n_k$ qui est remplie itérativement, case par case : calcul itératif

```
function P(n_1, ..., n_k)
M ← tableau(n_1, ..., n_k);
// remplir M avec les cas de base
for i_1 ← 1 to n_1 do
    ...;
    for i_k ← 1 to n_k do
        ...;
        M(i_1, ..., i_k) ← ...;
        // utiliser les valeurs déjà calculées
return M(n_1 ... n_k);
```

Complexité

- $\Theta(n_1 \times \cdots \times n_k \times \ell)$ opérations
- $\Theta(n_1 \times \cdots \times n_k)$ mémoire sur le tas
- $O(1)$ mémoire sur la pile
- Parfois trop d'opérations !

Mémoïsation vs programmation dynamique

- Les deux techniques reviennent essentiellement au même
- Avantage de la mémoïsation : seuls les calculs nécessaires sont effectués (on ne remplit pas complètement le tableau)
- Désavantage de la mémoïsation : les appels récursifs ont un petit surcoût (par rapport au style impératif) et, surtout, utilisent la pile d'appels (limitée en taille)
- On peut simuler le comportement de la programmation dynamique avec la mémoïsation
- Pour certains cas complexes, on peut simuler les appels récursifs avec une pile maintenue manuellement sur le tas

Reconstruire la solution

- Souvent utile de reconstruire une solution, pas juste de trouver une valeur optimale : meilleure combinaison de rendu de monnaie, plus court chemin, meilleur ensemble d'objets pour le sac à dos
- Comment faire ? Il suffit de se souvenir, quand on fait un min (ou un max), du choix ayant conduit au min (ou au max) – noter l'arg min en plus du min !
- En programmation récursive : renvoyer la sous-structure optimale, en plus de la valeur optimale
- Avec mémoïsation : noter dans la mémoire la sous-structure optimale, en plus de la valeur optimale
- En programmation dynamique : associer à chaque case du tableau la sous-structure optimale, en plus de la valeur optimale

Sac à dos fractionnel

- Variante du problème du sac à dos : on peut prendre des fractions d'objet

	o_1	o_2	o_3
poids	10	20	30
valeur	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

Sac à dos fractionnel

- Variante du problème du sac à dos : on peut prendre des fractions d'objet
- Retour sur la méthode gloutonne :

	o_1	o_2	o_3
poids	10	20	30
valeur	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

Sac à dos fractionnel

- Variante du problème du sac à dos : on peut prendre des fractions d'objet
- Retour sur la méthode gloutonne :
  - On trie les objets selon leur $\frac{v_i}{w_i}$ par ordre décroissant

	o_1	o_2	o_3
poids	10	20	30
valeur	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

Sac à dos fractionnel

- Variante du problème du sac à dos : on peut prendre des fractions d'objet
- Retour sur la méthode gloutonne :
  - On trie les objets selon leur $\frac{v_i}{w_i}$ par ordre décroissant
  - On prend les objets dans l'ordre trié, en prenant autant que possible de chaque objet

	o_1	o_2	o_3
poids	10	20	30
valeur	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

Sac à dos fractionnel

- Variante du problème du sac à dos : on peut prendre des fractions d'objet
- Retour sur la méthode gloutonne :
  - On trie les objets selon leur $\frac{v_i}{w_i}$ par ordre décroissant
  - On prend les objets dans l'ordre trié, en prenant autant que possible de chaque objet
- Dans le cas fractionnel, c'est optimal !

	o_1	o_2	o_3
poids	10	20	30
valeur	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

Sac à dos fractionnel

- Variante du problème du sac à dos : on peut prendre des fractions d'objet
- Retour sur la méthode gloutonne :
  - On trie les objets selon leur $\frac{v_i}{w_i}$ par ordre décroissant
  - On prend les objets dans l'ordre trié, en prenant autant que possible de chaque objet
- Dans le cas fractionnel, c'est optimal !
- Donne une complexité en $\Theta(n \log n)$, voir le cours sur les *Algorithmes de tri*

	o_1	o_2	o_3
poids	10	20	30
valeur	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

Cas d'application des algorithmes gloutons

- Quand peut-on et est-il intéressant d'utiliser un algorithme glouton ?

Cas d'application des algorithmes gloutons

- Quand peut-on et est-il intéressant d'utiliser un algorithme glouton ?
- On a besoin de deux conditions :

Cas d'application des algorithmes gloutons

- Quand peut-on et est-il intéressant d'utiliser un algorithme glouton ?
- On a besoin de deux conditions :

  Sous-structures optimales. Les solutions optimales des sous-problèmes sont des sous-structures de la solution optimale du problème orginal.

Cas d'application des algorithmes gloutons

- Quand peut-on et est-il intéressant d'utiliser un algorithme glouton ?
- On a besoin de deux conditions :

  Sous-structures optimales. Les solutions optimales des sous-problèmes sont des sous-structures de la solution optimale du problème orginal.

  Optimalité du choix glouton. Le choix optimal localement est un choix optimal globalement.

Plan

Références

- Programmation dynamique : chap. 15 de [Cormen et al., 2009, 2010]

Références

- Programmation dynamique : chap. 15 de [Cormen et al., 2009, 2010]
- Algorithmes gloutons : chap. 16 de [Cormen et al., 2009, 2010]

Bibliographie

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. ISBN 978-0-262-03384-8. URL http://mitpress.mit.edu/books/introduction-algorithms.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Algorithmique*. Dunod, 3rd edition, 2010. ISBN 978-2-100-54526-1. URL https://www.dunod.com/sciences-techniques/algorithmique-cours-avec-957-exercices-et-158-problemes.

Jake VanderPlas. *Python Data Science Handbook*. O'Reilly, 2016. https://jakevdp.github.io/PythonDataScienceHandbook/.

Ressources utilisées et licence

La représentation de la mémoire est due à ProgMan (Wikimedia), CC-BY-SA-4.0.
L'ensemble du contenu de cette présentation est sous licence CC-BY-4.0, à l'exception des éléments ci-dessus qui conservent une licence plus restrictive.