

Programmation dynamique

Denis Trystram

ENSIMAG alternants 2A

Octobre 2022

Présenter la méthode de **Programmation Dynamique** sous ses différents aspects : Conception, analyse (et implémentation).

- 1 Exemple introductif
- 2 Formalisation : Principe
- 3 Exemple : sac-à-dos

Bagdad, IX siècle

On rapporte l'histoire d'un chamelier qui négocia le prix d'une caravane avec un émissaire du calife El Maamun, fils du célèbre Harun al Rashid ^a.

a. Giegerich et al. Science of Computer Programming, 2004

- L'affaire fut conclue, la formule inscrite sur le sable :
 $(1 + 2) \times 3 \times 4 + 5 = 41$ barils d'huile.
- Le temps d'une prière, le vent efface les parenthèses :
 $1 + 2 \times 3 \times 4 + 5$.
- Arrivé au palais, il a fallu recalculer la valeur.
Le chamelier proposa (évidemment)
 $(1 + 2) \times 3 \times (4 + 5) = 81$.

Résolution "à la main"

Al Khwarizmi fut appelé au palais pour tirer au clair cette affaire. Il trouva tout d'abord qu'il y avait 14 façons différentes de parenthéser l'expression.

- $(1 + 2) \times 3 \times (4 + 5) = 81$
- $1 + 2 \times (3 \times 4 + 5) = 35$
- $(1 + 2 \times 3) \times 4 + 5 = 33$
- ...

Bien sûr, il est possible de calculer toutes les valeurs et d'en prendre le minimum (valeur qui avantageait le Calife), mais Al Khwarizmi proposa une méthode générale^a.

Pour cela, il reçut une bourse d'étude.

a. Il n'est pas le "père de l'algorithmique" sans raison !



Formalisation du problème

On note τ l'expression complète (la chaîne de caractères) :

$$\tau = 1 + 2 \times 3 \times 4 + 5$$

et on indice les caractères selon leur place.

$$\tau = {}_0 1_1 + {}_2 2_3 \times {}_4 3_5 \times {}_6 4_7 + {}_8 5_9$$

Ainsi, $t(i, j)$ désigne la sous-expression entre i et j .

Par exemple, $t(2, 5) = 2 \times 3$.

La *meilleure* valeur (minimale pour le Calife) de la sous-expression est stockée dans un tableau (T).

L'objectif est de déterminer $T(0, 9)$.

On définit

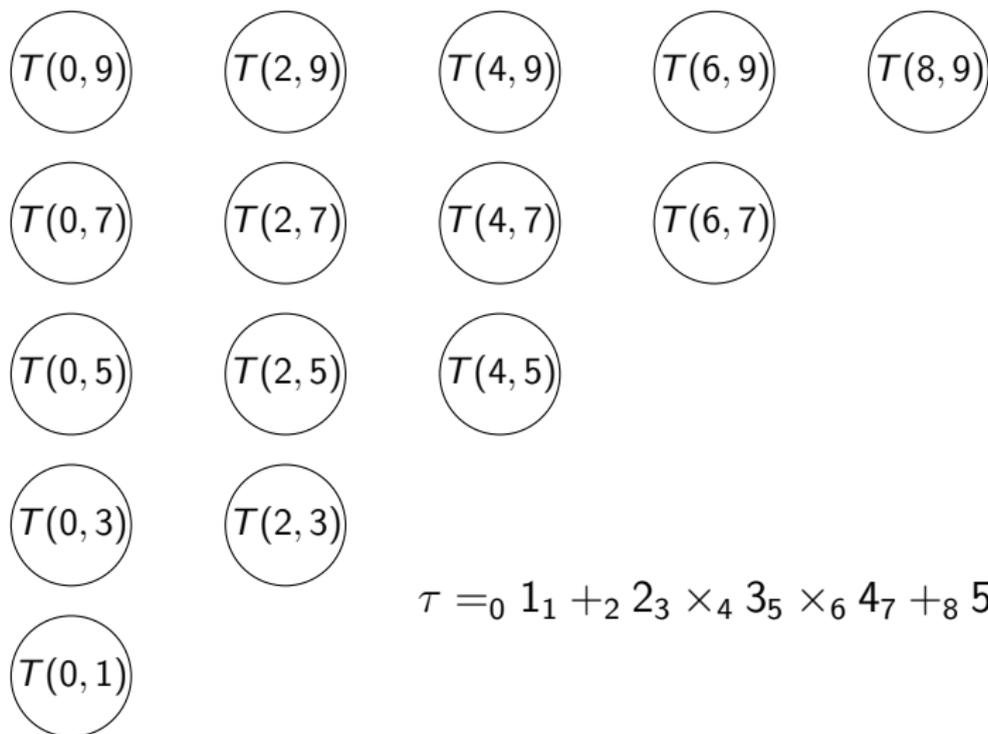
- $T(i, i + 1) = n$ si $t(i, i + 1) = n$
- $T(i, j) = \min_{k|i < k < j} [T(i, k)op(k, k + 1)T(k + 1, j)]$

Ainsi, partant des petites valeurs de $(i, i + 1)$, on peut calculer successivement toutes les entrées de la table T .

$$T(0, 9) = \min[T(0, 1) + T(2, 9), T(0, 3) \times T(4, 9), T(0, 5) \times T(6, 9), T(0, 7) + T(8, 9)].$$

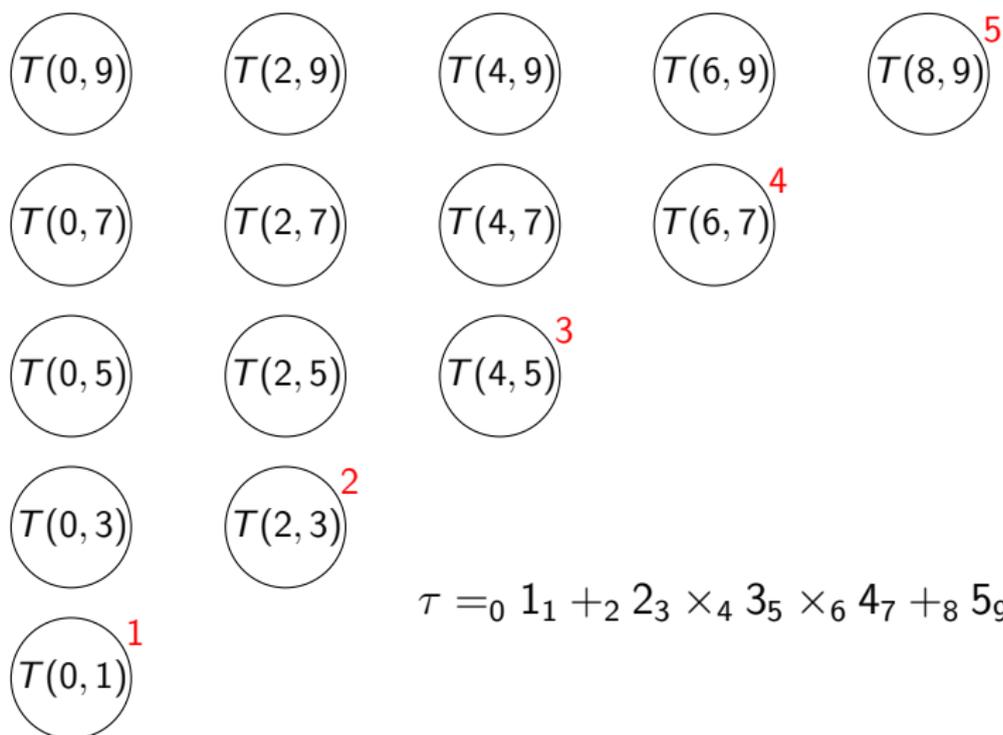
Comment faire ?

- On part de ce que l'on connaît...
- Les $T(i, i + 1) = n$ si $t(i, i + 1) = n$
 $T(0, 1), T(2, 3), T(4, 5), T(6, 7), T(8, 9)$
- On en déduit de proche en proche toutes les autres valeurs de T

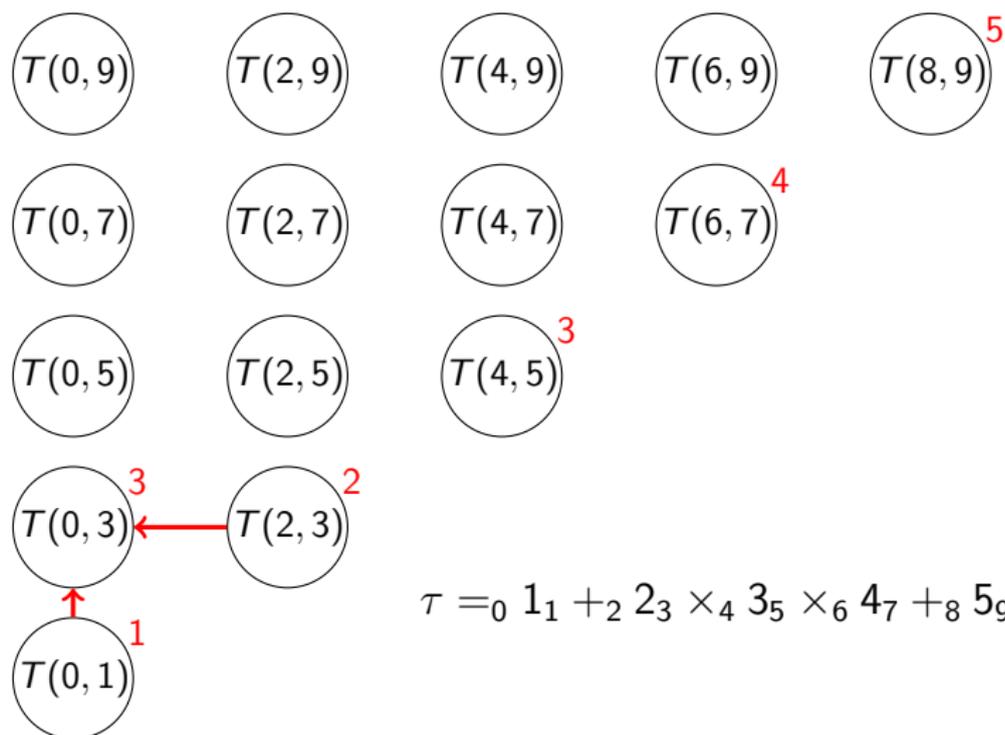


$$\tau = 0 \ 1_1 + 2 \ 2_3 \times 4 \ 3_5 \times 6 \ 4_7 + 8 \ 5_9$$

Déroulement

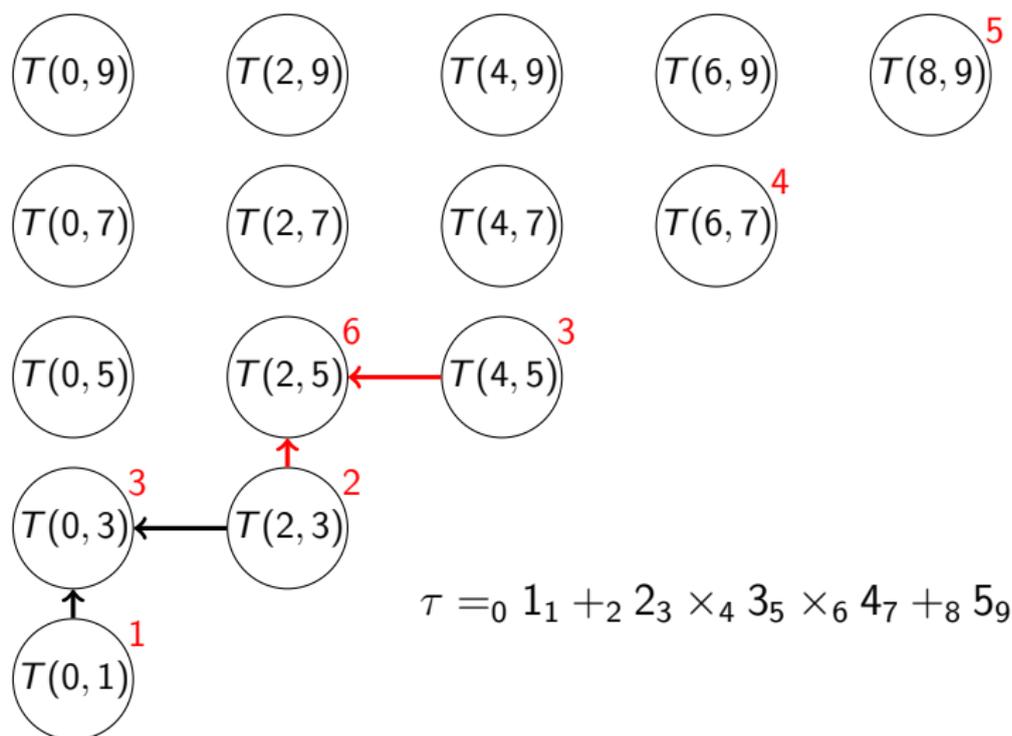


Déroulement

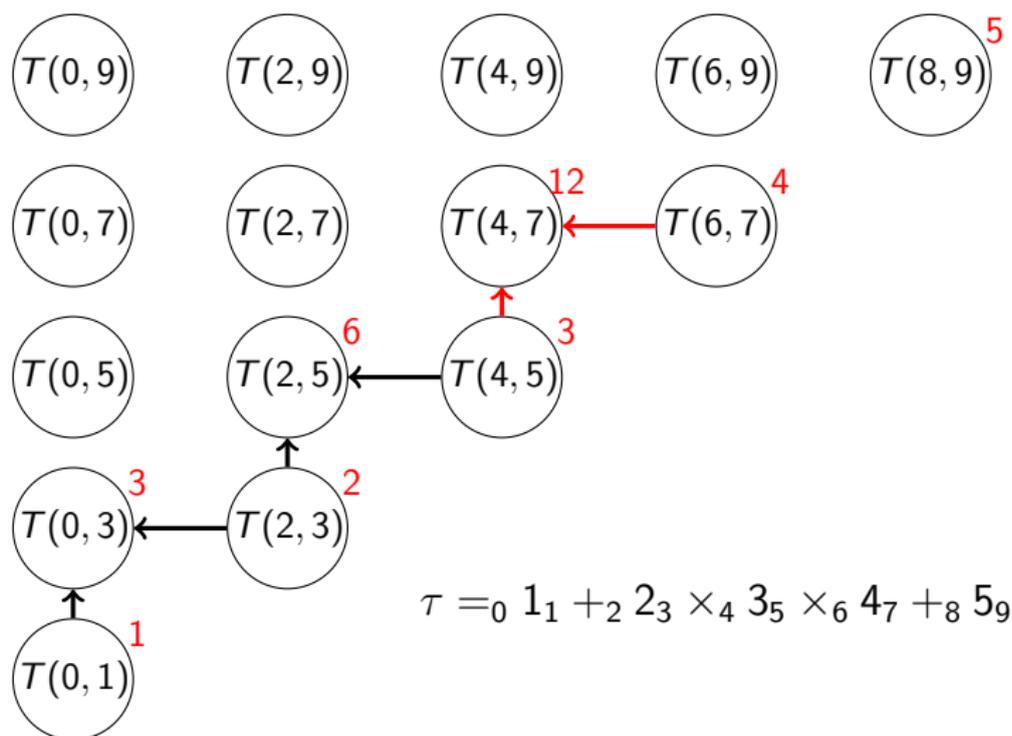


$$\tau = 0 \ 1_1 + 2 \ 2_3 \times 4 \ 3_5 \times 6 \ 4_7 + 8 \ 5_9$$

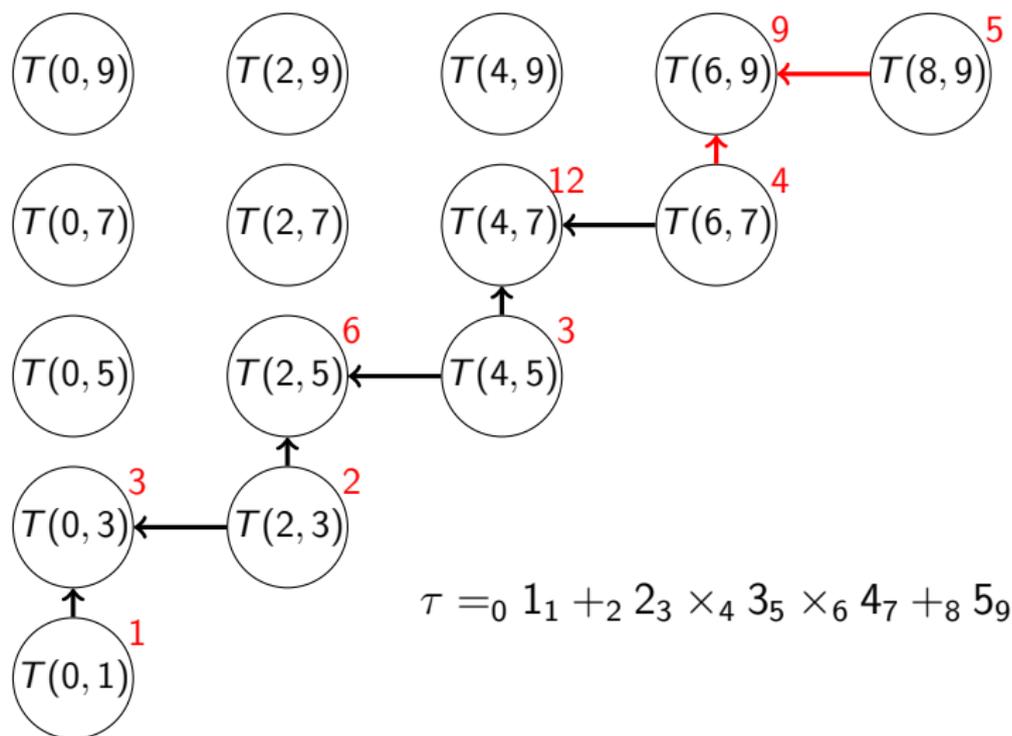
Déroulement



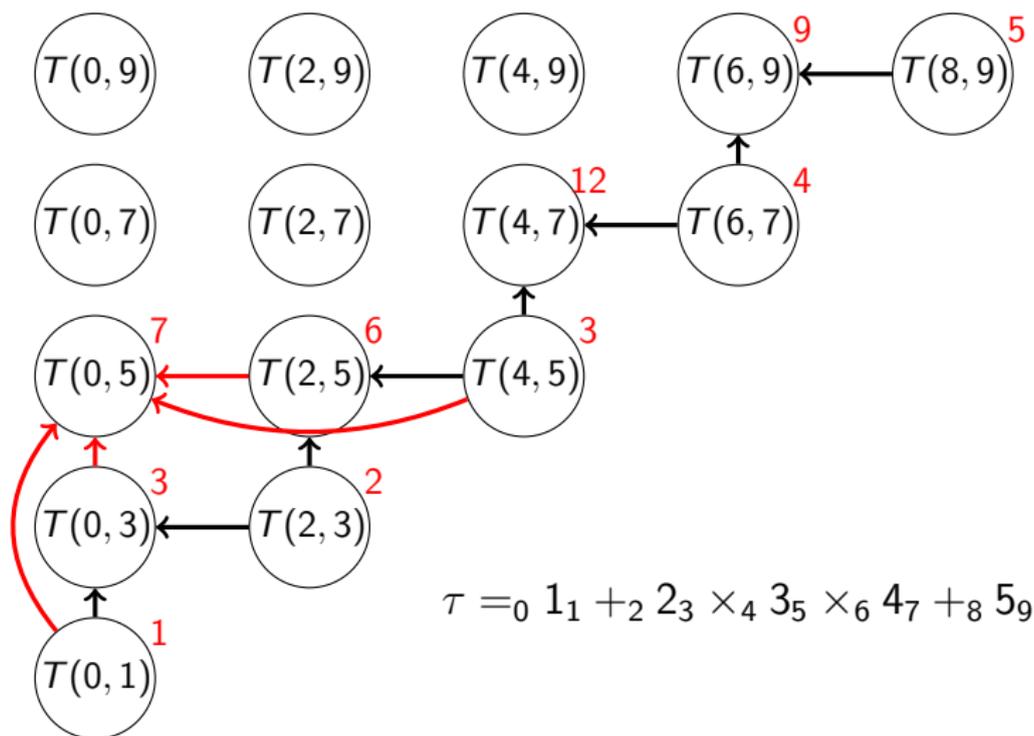
Déroulement



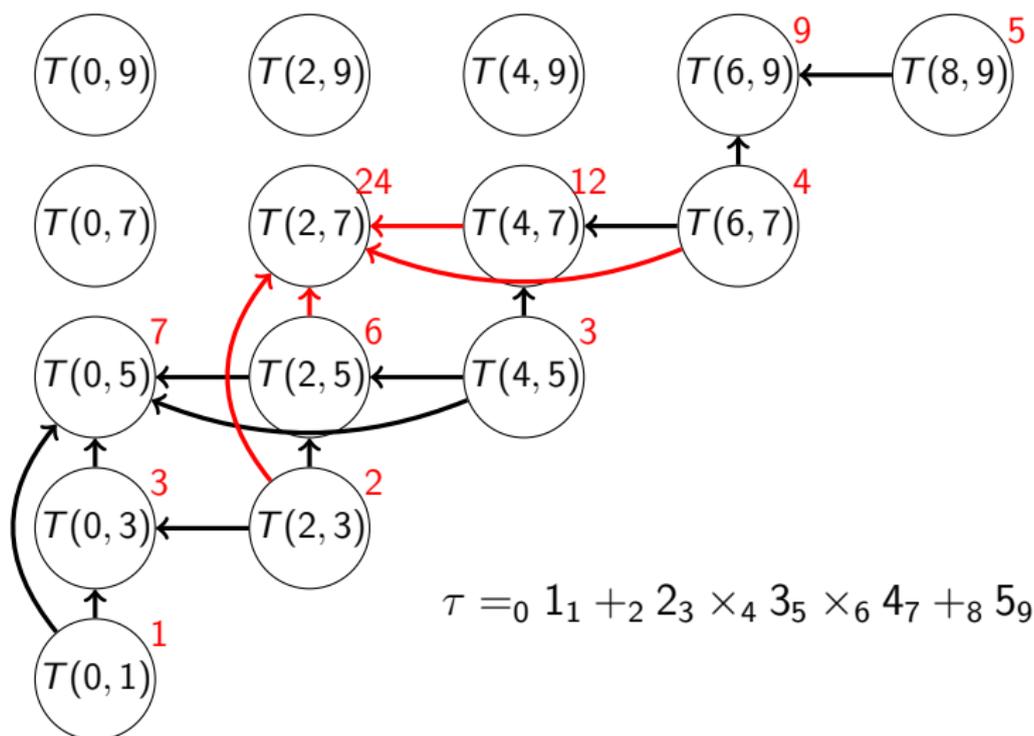
Déroulement



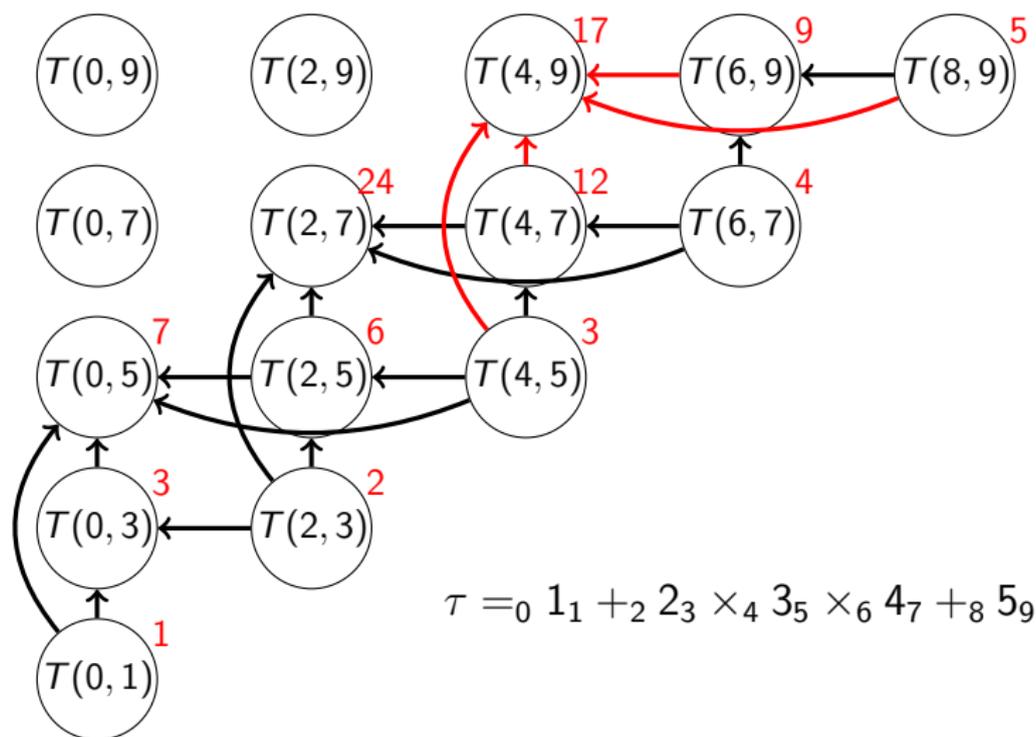
Déroulement



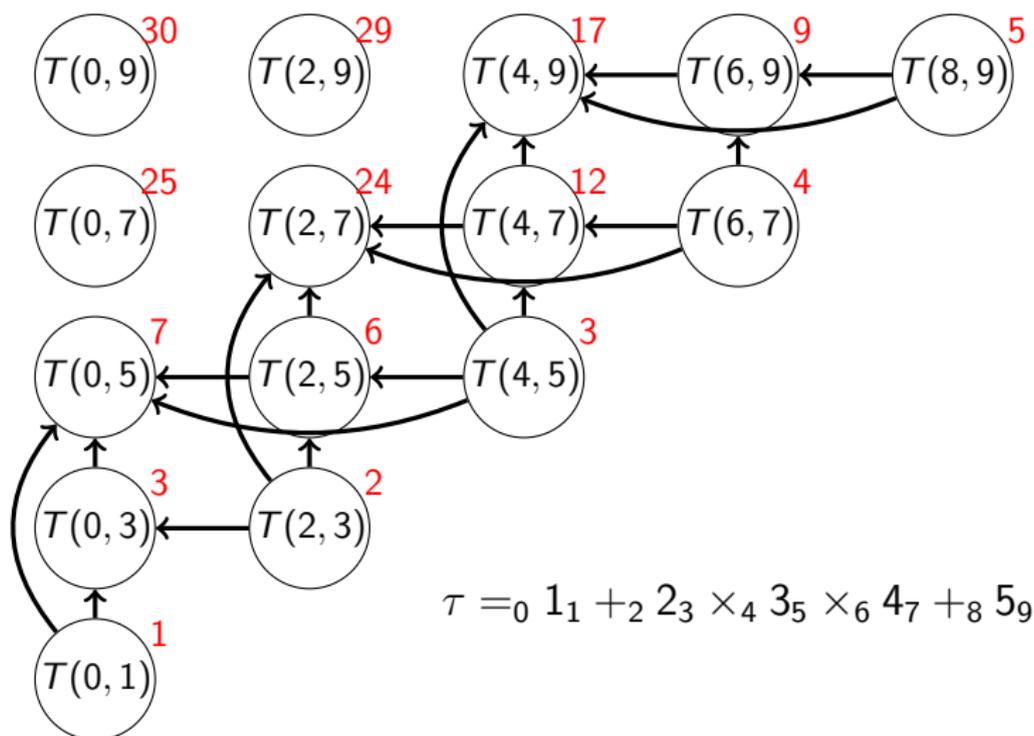
Déroulement



Déroulement



Déroulement



Encore plus fort !

- On peut rajouter une variable pour tracer le chemin (et reconstruire comment la solution a été obtenue).
- On peut calculer de la même façon la valeur maximale¹.
Il suffit de changer le min en max dans les expressions.
On trouve $T'(0, 9) = 81$.
On notera ici que le chemin pour obtenir la solution n'est pas le même que dans le cas du minimum.

1. celle qu'a fournie le chamelier et qui lui a coûté la tête ! 

- 1 Exemple introductif
- 2 Formalisation : Principe
- 3 Exemple : sac-à-dos

Définition

La **programmation dynamique** est une méthode de décomposition d'un problème d'optimisation en sous-problèmes telle que la solution soit obtenue "facilement" dès que l'on connaît celles de tous les sous-problèmes.

La difficulté ici est de déterminer les bons sous-problèmes.

Principe de (sous-optimalité) de Bellman² : La solution d'un problème d'optimisation peut être obtenue par combinaisons de solutions optimales sur les sous-problèmes.

2. livre de Bellman "Dynamic Programming", Princeton en 1953

Tiré de l'autobiographie de Bellman "Eye of the hurricane" en 1984 (réédité).

- Bellman travaillait à l'époque dans une entreprise aux US (RAND), il raconte qu'il a choisi "programming" pour ne pas effrayer son chef et parce que cela lui rappelait "Programmation Linéaire".
- Le terme "dynamic" vient de l'aspect de l'évolution de la méthode dans la remontée des différents étages de la récurrence.

Cela ne vous rappelle rien ?

Attention

c'est différent du classique *divide-and-conquer* où il faut TOUT générer.
Exemples : TriFusion, enveloppe convexe, Karatzuba, ...

De plus, cette technique impose que les sous-problèmes soient disjoints (partition).

L'interprétation du principe de sous-optimalité de Bellman :

- comme les opérations $+$ et \times sont monotones sur les entiers positifs, les expressions $x + y$ et $x \times y$ sont minimales (resp. maximales) lorsque x et y sont évaluées à leurs valeurs minimales (resp. maximales).

- 1 Exemple introductif
- 2 Formalisation : Principe
- 3 Exemple : sac-à-dos**

Définition du problème

On considère n objets ayant chacun un poids w_i et une valeur v_i ($1 \leq i \leq n$) et un sac-à-dos S de capacité W .

On veut maximiser la valeur totale ($\sum_{i \in S} v_i$) sous la contrainte $\sum_{i \in S} w_i \leq W$.

Exemple (1)

On considère l'instance suivante de 6 objets pour un sac de capacité 11 :

index	1	2	3	4	5	6
	(2,5)	(3,7)	(5,16)	(4,13)	(7,19)	(3,10)

Exemple (2)

Résolution par un algorithme **glouton** :

- On trie les objets par leurs rapports *qualité-prix* : 6, 4, 3, 5, 1, 2
- on affecte les objets dans cet ordre tant qu'il y a de la place dans le sac.

Solution : 3 objets (6, 4 et 1) pour une valeur totale de 28.

Double récurrence.

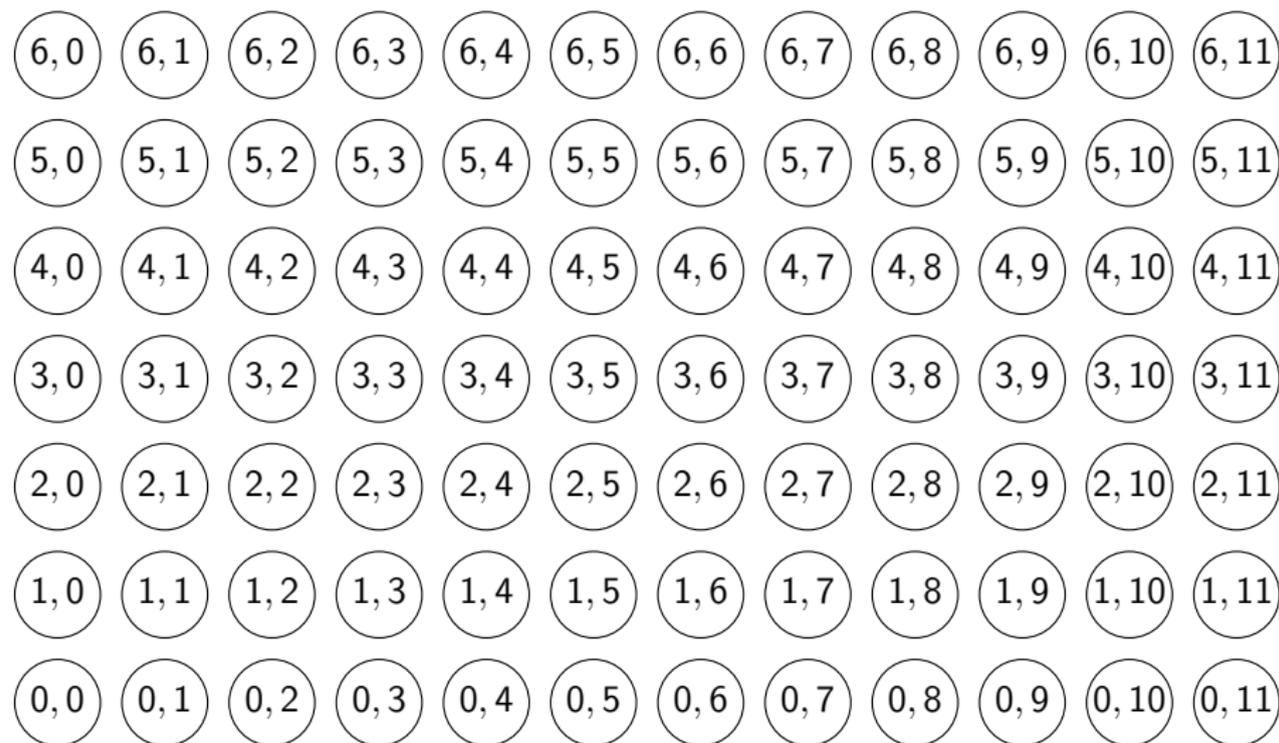
On note $V(k, p)$ la meilleure valeur totale des k premiers objets dans un sac de capacité p .

On initialise à 0.

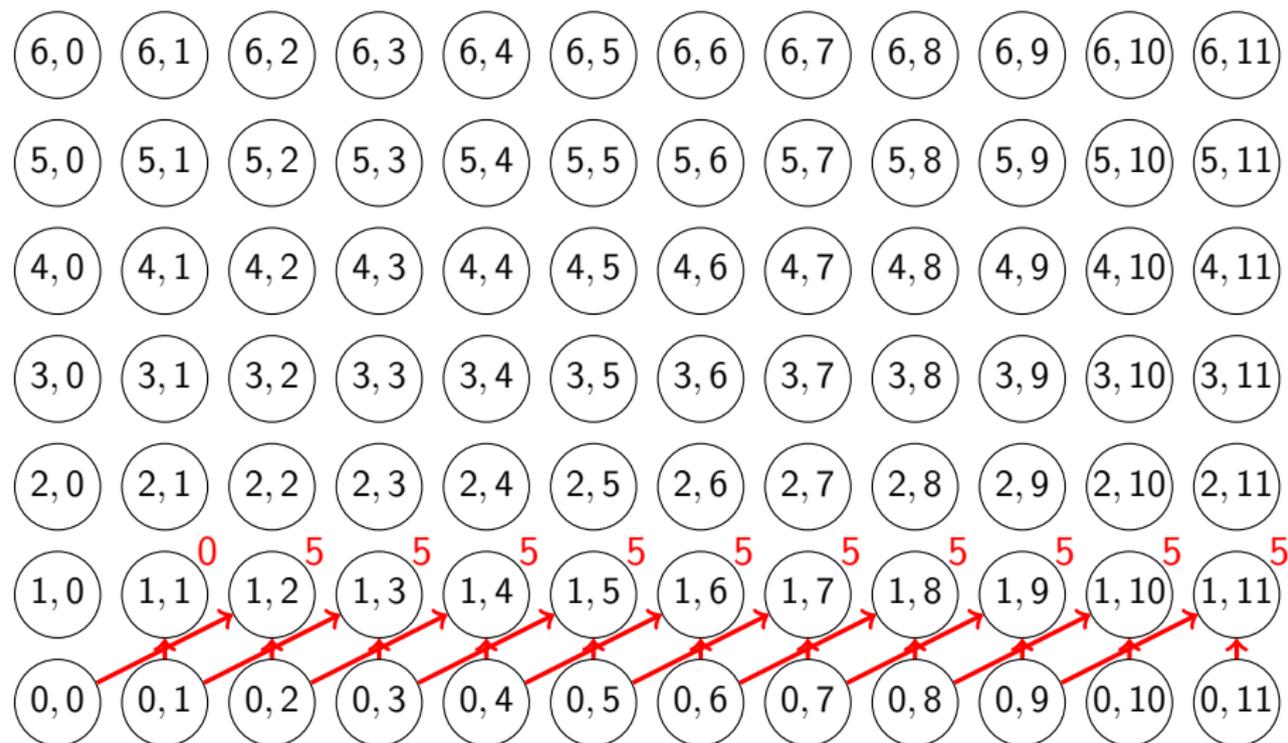
- $V(1, p) =$
 - 0 si $p < w_1$
 - v_1 si $p \geq w_1$
- Pour les autres objets $k > 1$,
 $V(k, p) =$
 - $V(k - 1, p)$ si $p < w_k$
 - $\max[V(k - 1, p), V(k - 1, p - w_k) + v_k]$ si $p \geq w_k$

La solution est $V(n, W)$

Résolution



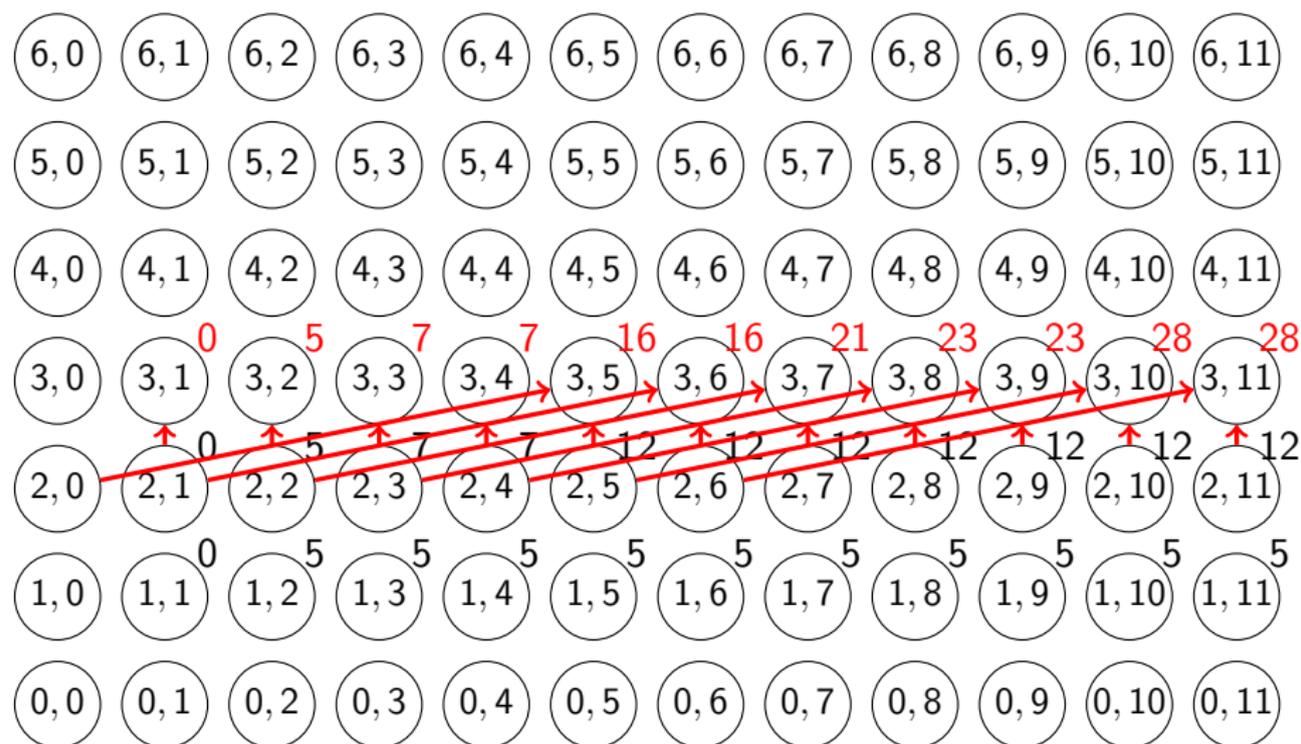
Résolution



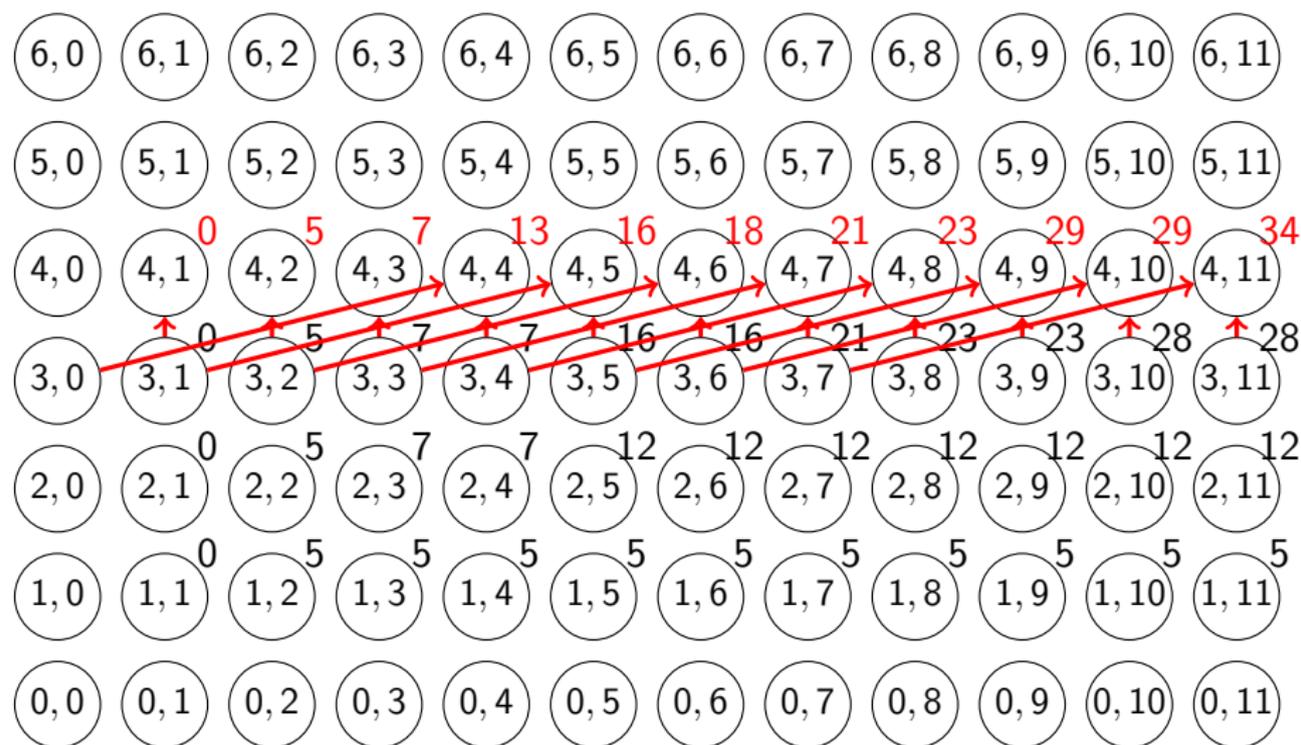
Résolution



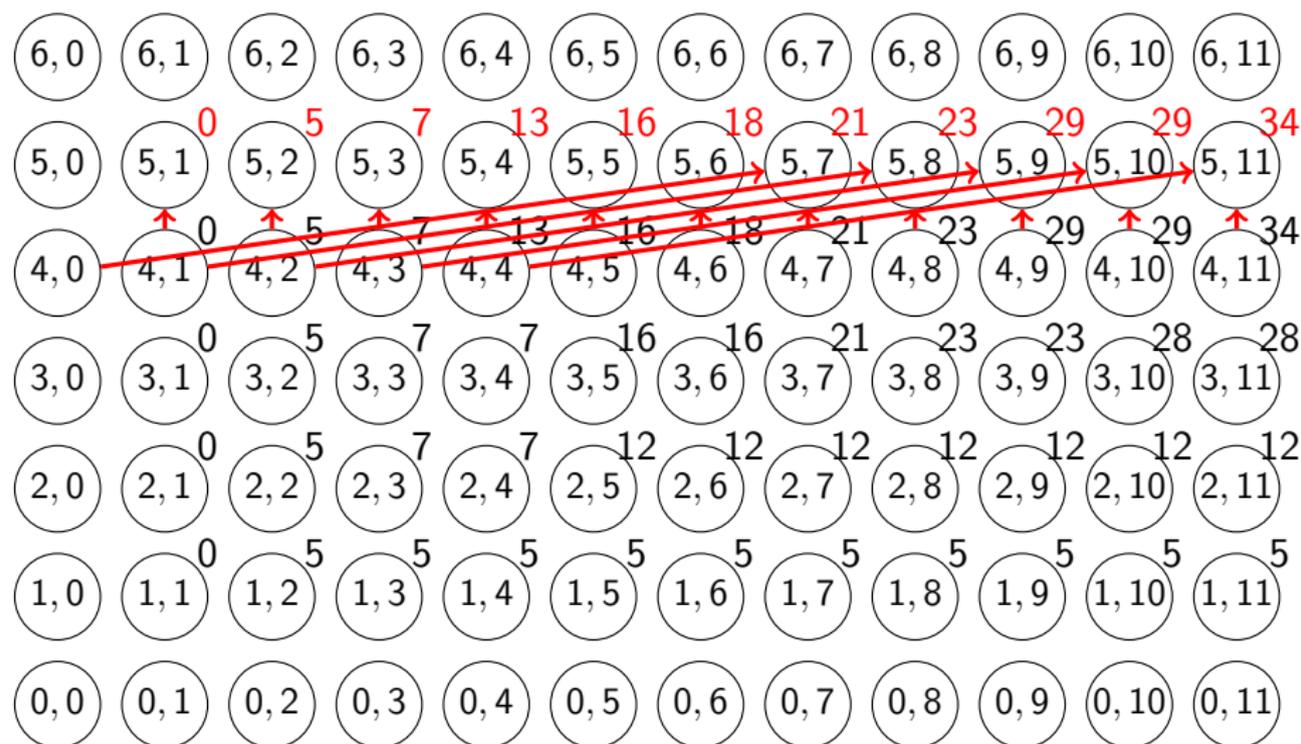
Résolution



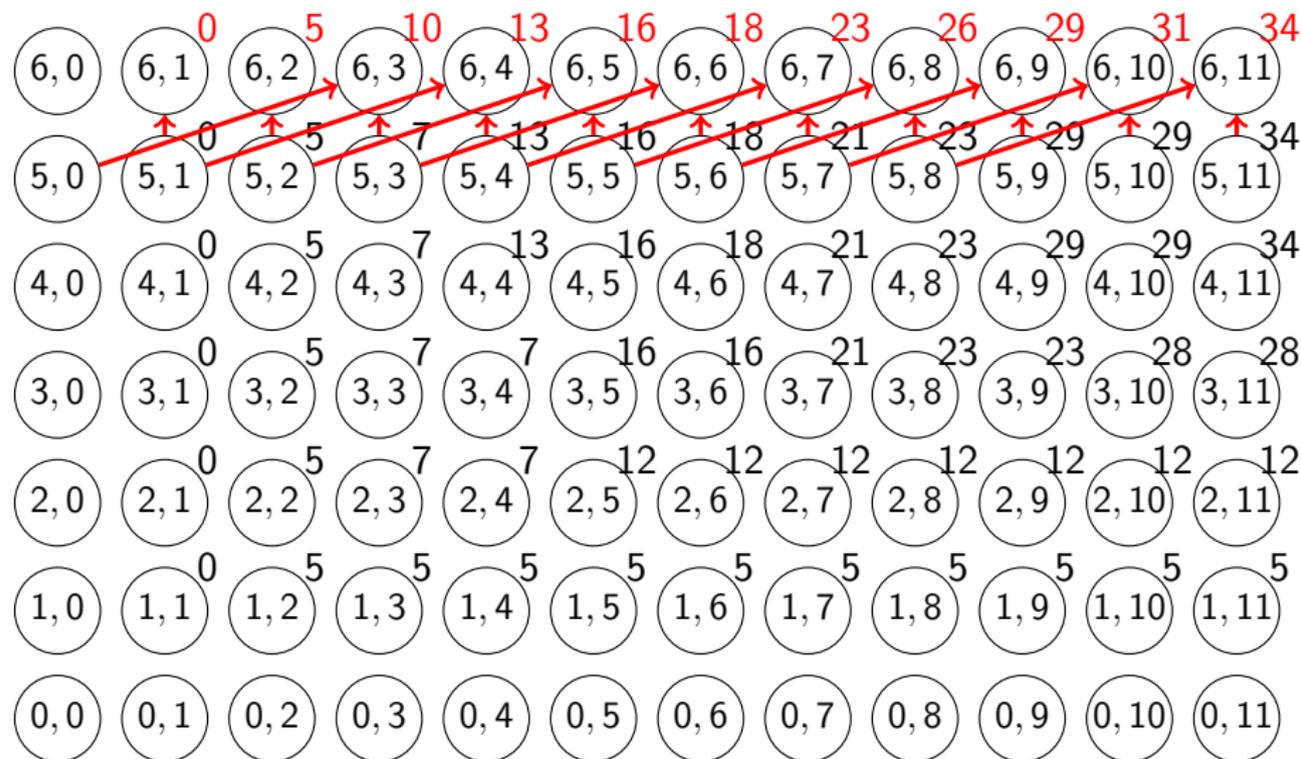
Résolution



Résolution



Résolution



- On peut retrouver la solution si on a marqué le chemin en remontant dans le tableau.
- Le temps d'exécution de l'algorithme précédent pour résoudre le problème du sac-à-dos est dans $O(n \times W)$
il n'est pas seulement polynomial en n mais dépend du plus grand entier en entrée du problème (W).
- Le problème est NP-difficile, mais il est *pseudo-polynomial* :
Il peut être résolu efficacement quand les nombres w_i sont petits.