

# **ALGORITHMES POUR MAXIMISER LES BÉNÉFICES**

Aide à la décision dans le processus d'achat d'actions

Plan

1 Problème

2 Algorithmes

3 Résultats

4 Conclusion

# Problème

- ❖ Comment maximiser le profit total des actions achetées, sous les contraintes :
    - Chaque action : **achetée qu'une seule fois.**
    - Acheter une fraction d'action : **non**
    - Dépense maximal : **500 euros**
- => Problème du sac à dos (the 0/1 Knapsack Problem)

# Algorithmes

- Les algorithmes pour résoudre le problème :
  - Brute Force
  - Dynamic Programming
  - Greedy
  - Etc.

# Représenter le problème

- Étant donnée n actions :  $A_1, A_2, \dots, A_n$
- Noter  $C_i$  le coût de  $A_i$
- Noter  $P_i$  le profit de l'action  $A_i$   
=> Le *profit total* de  $A_i$  :  $C_i * P_i$   
(Si  $P_i$  représente en « % », le *profit total* de  $A_i$  :  $C_i * P_i * 0.01$ )
- Le problème est de trouver :  
(ici,  $C_{\max} = 500$ )

$$\text{Max}(\sum_i C_i * P_i) \quad \text{sous la contrainte} \quad \sum C_i \leq C_{\max}$$

# Algorithme Brute Force

- Lister toutes les différentes combinaisons d'actions
- Pour chaque combinaison, calculer le profit total
- La solution est la combinaison ayant le profit total le plus grand et le coût total ne dépasse pas  $C_{\max}$

# Brute Force - Exemple avec 3 actions

- Les combinaisons (lire l'arbre de gauche à droite):

1. A1A2A3

2. A1A2

3. A1 A3

4. A1

5. A2A3

6. A2

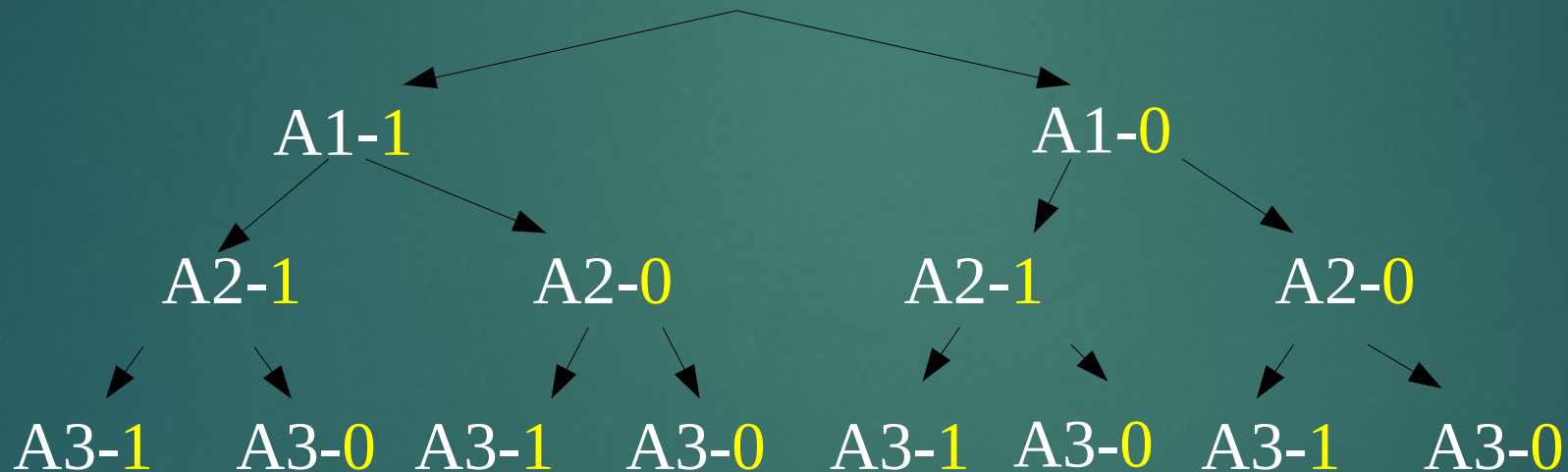
7. A3

8. 0

$\Rightarrow 2^3 = 8$  combinaisons

Chaque combinaison = représentation binaire d'un nombre de 0 à  $2^3 - 1$   
 $\Leftrightarrow x_1x_2x_3$  où  $x_1, x_2, x_3$  choisi dans  $\{0, 1\}$

Exemple en binaire: 111, 110, ..., 011 (=11), ..., 001 (=1), 000 (=0)



Ai-1 : Ai est choisi

Ai-0 : Ai n'est pas choisi

L'arbre représente les choix dans l'algorithme Brute Force

# Brute Force - Pseudocode

# Complexité

Mémoire / Temporelle  
 $O(n)$   $O(n*2^n)$

n = nombre\_de\_actions

nombre\_de\_combinaisons =  $2^n$

solution = []

=> *n éléments*

Pour k de 0 à nombre\_de\_combinaisons – 1 :

=>  *$2^n$  opérateurs*

    choix = représenter k en binaire (en n chiffres binaires)

    combinaison = []

=> *n éléments*

        Pour chaque élément-i dans le choix :

=> *n opérateurs*

            Si élément-i == 1 :

                action-i est choisie donc l'ajouter à la combinaison

Si le coût total de la combinaison  $\leq C_{\max}$  :

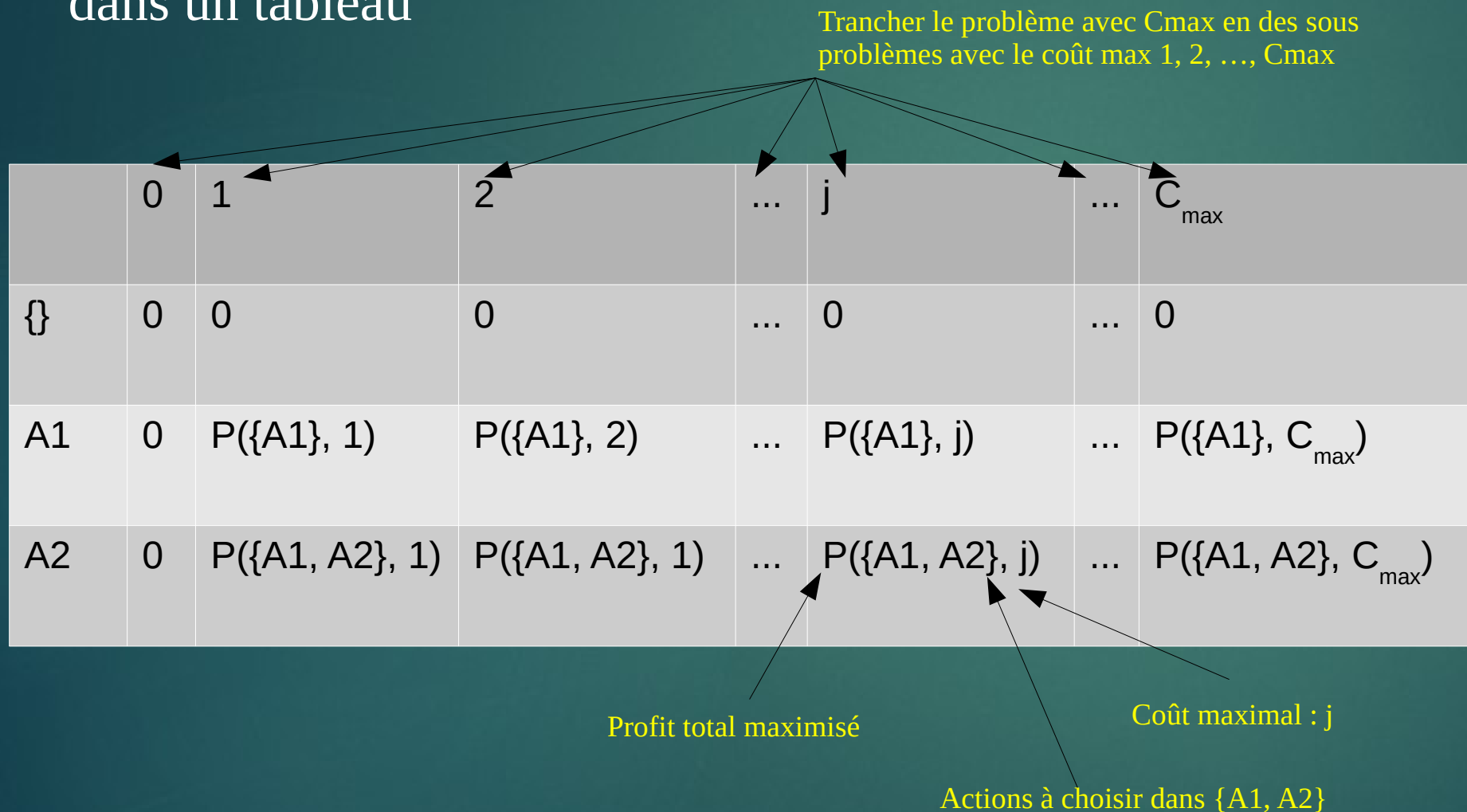
    Si le profit total de la combinaison > le profit total de la solution :

        solution = combinaison



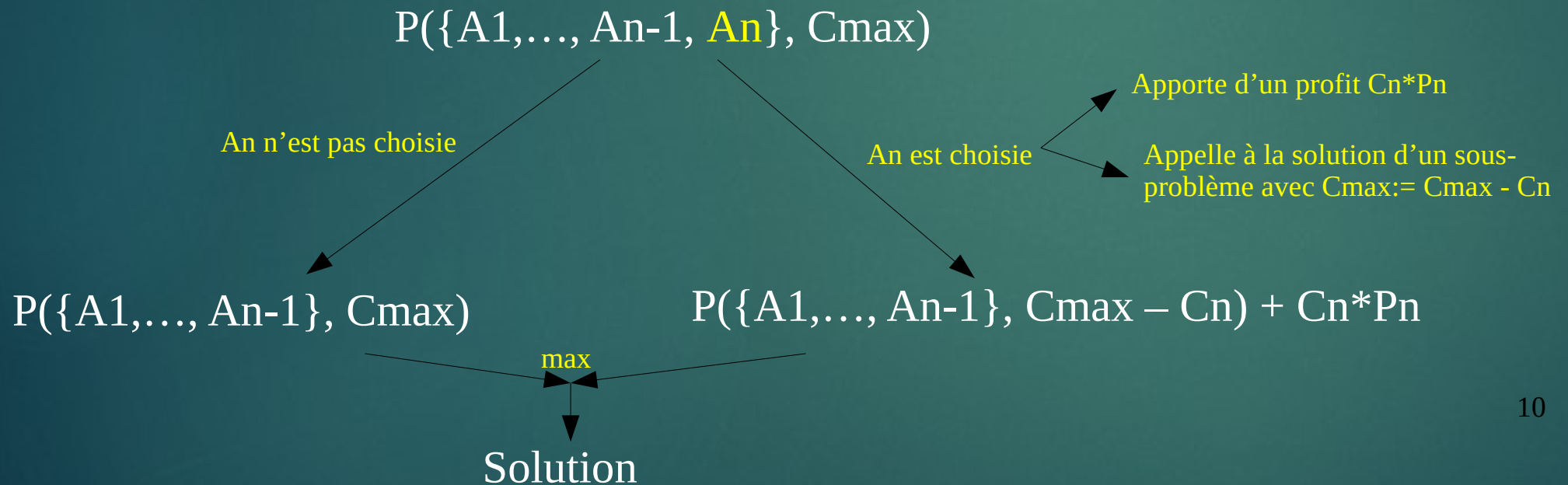
# Algorithme Programmation dynamique

- Construire la solution optimale du problème à  $i$  actions à partir du problème à  $i-1$  actions (les actions sont numérotés  $1, 2, \dots, n$ )
- Les solutions de sous-problèmes sont pré-calculées et stockées dans un tableau



# Algorithme Programmation dynamique (suite)

	0	...	...	$C_{\max}$
...	...		...	0
$A_{n-1}$	...	$P(\{A_1, \dots, A_{n-1}\}, C_{\max} - C_n)$	...	$P(\{A_1, \dots, A_{n-1}\}, C_{\max})$
$A_n$	...	...	...	$P(\{A_1, \dots, A_{n-1}, A_n\}, C_{\max})$



# Programmation dynamique

## Pseudocode

## Complexité

Mémoire / Temporelle  
 $O(n*m)$   $O(n*m)$

$n = \text{nombre\_de\_actions} + 1$

$m = \text{int}(C \text{ max} ) + 1$

B = matrice de taille  $n*m$  initialisée avec 0

=>  $n*m$  éléments

pour stocker les profits totales optimales de sous problèmes

# remplir la matrice B

Pour i de 1 à n :

=>  $n*m$

Pour j de 1 à m :

opérateurs

Si coût(action-i) < j :

l'action-i est ajoutée dans la liste des actions à choisir donc

$B[i][j] = \max(B[i-1][j], B[i-1][j - \text{coût}(\text{action-i})] + \text{profit}(\text{action-i}))$

Sinon :

$B[i][j] = B[i-1][j]$

solution = []

~  $n$  éléments

# tracer la solution

Tant que  $C \text{ max} \geq 0$  et  $n \geq 0$  :

Si  $B[n][C \text{ max} ] == B[n-1][C \text{ max} - \text{coût}(\text{action-n})] + \text{profit}(\text{action-n})$  :

~  $n*m$  opérateurs

l'action-n est choisie donc l'ajouter dans la solution

$C \text{ max} = C \text{ max} - \text{coût}(\text{action-n})$

$n = n - 1$

# Algorithme Greedy

- Trier les actions en fonction de leurs efficacités (le rapport de son profit sur son coût).
- Ajouter les actions les plus efficaces dans la limite du coût maximal autorisé.

# Algorithme Greedy - Exemple

	Coût (euros)	Profit (euros)	Efficacité (Profit/Coût)
Action-1	10	1	$1/10 = 0.1$
Action-2	15	6	$6/15 = 0.4$
Action-3	5	1	$1/5 = 0.2$

*Trier en efficacité :*

*=> 3ème*

*=> 1er*

*=> 2ème*

- Liste d'actions triées : [Action-2, Action-3, Action-1]
- Si  $C_{\max} = 20$ , l'action 2 est choisie car  $C_2 = 15 < 20$
- Si  $C_{\max} = 28$ , l'action 2 est choisie, puis l'action 3 car  $C_2 + C_3 = 15 + 5 < 28$
- Si  $C_{\max} = 31$ , l'action 2 est choisie, puis l'action 3, enfin l'action 1 car  $C_2 + C_3 + C_1 = 15 + 5 + 10 < 31$

# Greedy - Pseudocode

# Complexité

*Mémoire / Temporelle*  
 *$O(n)$      $O(n \log(n))$*

$n = \text{nombre\_de\_actions}$

$L = \text{liste des actions triées en ordre décroissant de l'efficacité}$

$\text{total\_coût} = 0$

$i = 1$

$\text{solution} = []$

Tant que  $i \leq n$  et  $\text{total\_coût} \leq C$  max :

$\text{action-}i = L[i]$

$\text{total\_coût} = \text{total\_coût} + \text{coût}(\text{action-}i)$

$i = i+1$

    ajouter action- $i$  à la solution

*$n$  éléments*

*Pour le trie :  $O(n \log(n))$*

*$\Rightarrow n$  éléments*

*$\Rightarrow n$  opérateurs  $\Rightarrow O(n)$*

# Résumé

	Complexité en mémoire	Complexité temporelle	Remarque
Brute Force	$O(n)$	$O(n \cdot 2^n)$	Temps de calcul exponentiel
Programmation dynamique	$O(n \cdot m)$	$O(n \cdot m)$	<ul style="list-style-type: none"><li>- Temps de calcul important si <math>m</math> est grand</li><li>- Coût total autorisé et le coût de chaque action doivent être un nombre entier</li></ul>
Greedy	$O(n)$	$O(n \log(n))$	<ul style="list-style-type: none"><li>- Très rapide</li><li>- Solution optimale locale</li></ul>

# Résumé en chiffres

	Complexité Temporelle ( $n = 20$ , $m = 50000$ )
Brute Force	$n \cdot 2^n = 20.971.520$
Programmation dynamique	$n \cdot m = 1.000.000$
Greedy	$n \log(n) = 26$



# Résultats

	Problème 20 actions	Problème dataset1 (956 actions)	Problème dataset2 (540 actions)
Brute Force	4 seconds  Total cost: 498 Total profit: 99.08		
Programmation dynamique	0.008 seconds Total cost: 498 Total profit: 99.08	27.87 seconds Total cost: 499.95 Total profit: 198.54	16.33 seconds Total cost: 499.90 Total profit: 197.96
Greedy	4.28e-05 seconds Total cost: 498 Total profit: 97.48	0.0006 seconds Total cost: 500.00 Total profit: 198.51	0.00045 seconds Total cost: 499.96 Total profit: 197.76
Données de Sienna		Total cost: 498.76 Total return: 196.61	Total cost: 489.24 Profit: 193.78

# Conclusion

- ❖ Algorithme brute force donne une solution exacte mais le temps de calcul est exponentiel en fonction du nombre d'actions.
- ❖ Algorithme programmation dynamique donne aussi une solution exacte mais le temps de calcul dépend du coût maximal autorisé (en entier) et devient très important lors que ce coût est grand.
- ❖ Algorithme greedy donne une solution approximative mais très rapide en temps de calcul. Cet algorithme est applicable quand le prix de chaque action est petit par rapport au coût maximal autorisé. Il peut donner une très mauvaise solution si l'action la plus efficace a le coût très proche du coût maximal autorisé.