ALGORITHMES POUR MAXIMISER LES BÉNÉFICES

Aide à la décision dans le processus d’achats d’actions

Plan

1. Problème
2. Algorithmes
3. Comparaison des 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 : A1, A2, …, An
- Noter Ci le coût de Ai
- Noter Pi le profit de la action Ai
  => Le *profit total* de Ai : Ci*Pi
  (Si Pi représente en « % », le *profit total* de Ai : Ci*Pi*0.01)
- Le problème est de trouver :
  (ici, C_{max} = 500)

$$Max(\sum_{i} C_i * P_i) \quad sous\ la\ constrainte \sum C_i \leq C_{max}$$

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

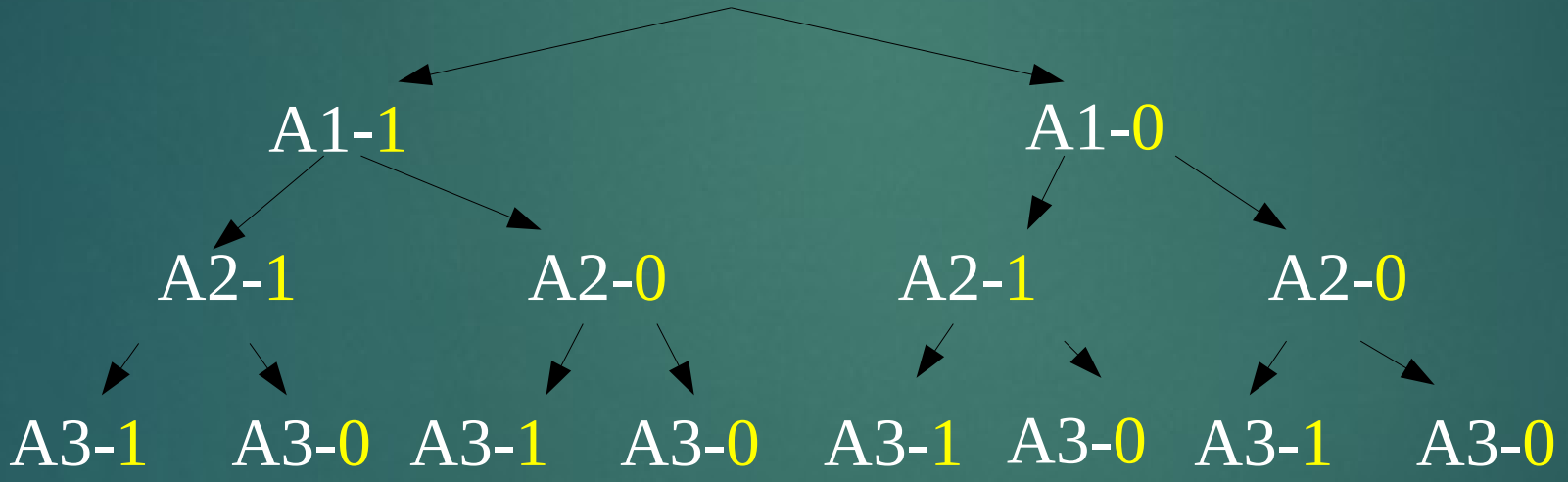
- Les combinaisons (lire l'arbre de gauche à droite):
  1. A1A2A3
  2. A1A2
  3. A1 A3
  4. A1
  5. A2A3
  6. A2
  7. A3
  8. 0

=> 2^3 = 8 combinaisons
Chaque combinaison = représentation binaire d'un nombre de 0 à $2^3 - 1$
<=> x1x2x3 où x1, x2, x3 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 élémént-i == 1 :
            action-i est choisie donc l'ajouter à la combinaison

    Si le coût total de la combinaison ≤ 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 numérotées 1, 2, …, n)

- Les solutions de sous-problèmes sont pré-calculées et stockées dans un tableau

Trancher le problème avec Cmax en des sous problèmes avec le coût max 1, 2, …, Cmax

	1	2	...	j	...	C_{max}
0	0	0	...	0	...	0
A1	P({A1}, 1)	P({A1}, 2)	...	P({A1}, j)	...	P({A1}, C_{max})
A2	P({A1, A2}, 1)	P({A1, A2}, 1)	...	P({A1, A2}, j)	...	P({A1, A2}, C_{max})

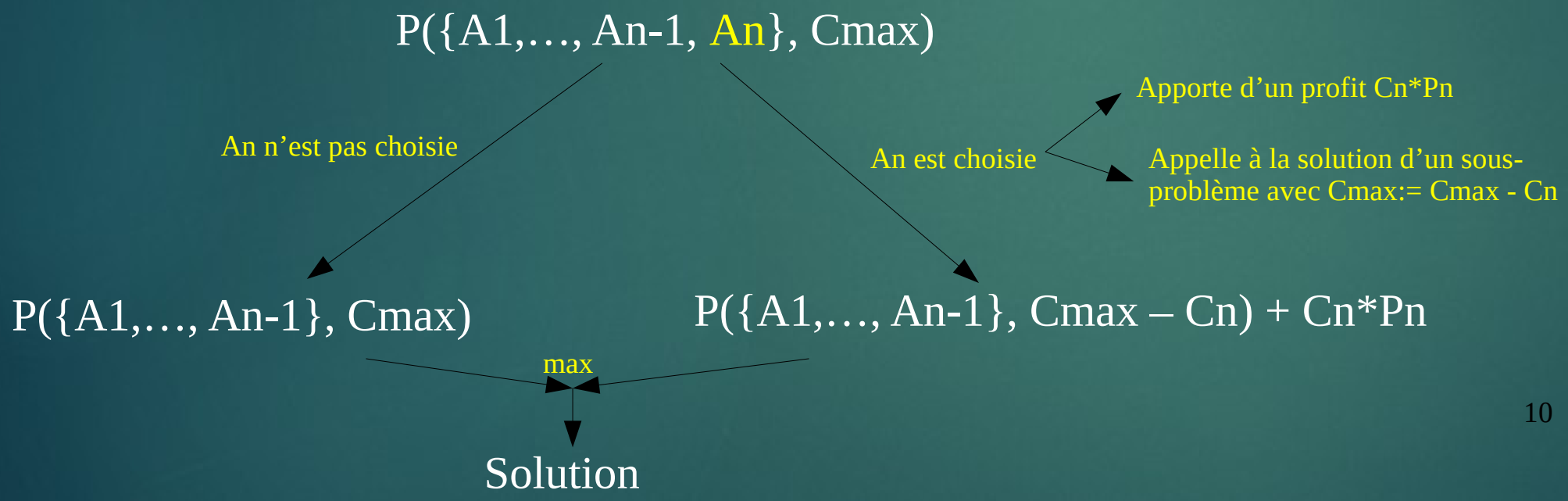
Profit total maximisé

Coût maximal : j

Actions à choisir dans {A1, A2}

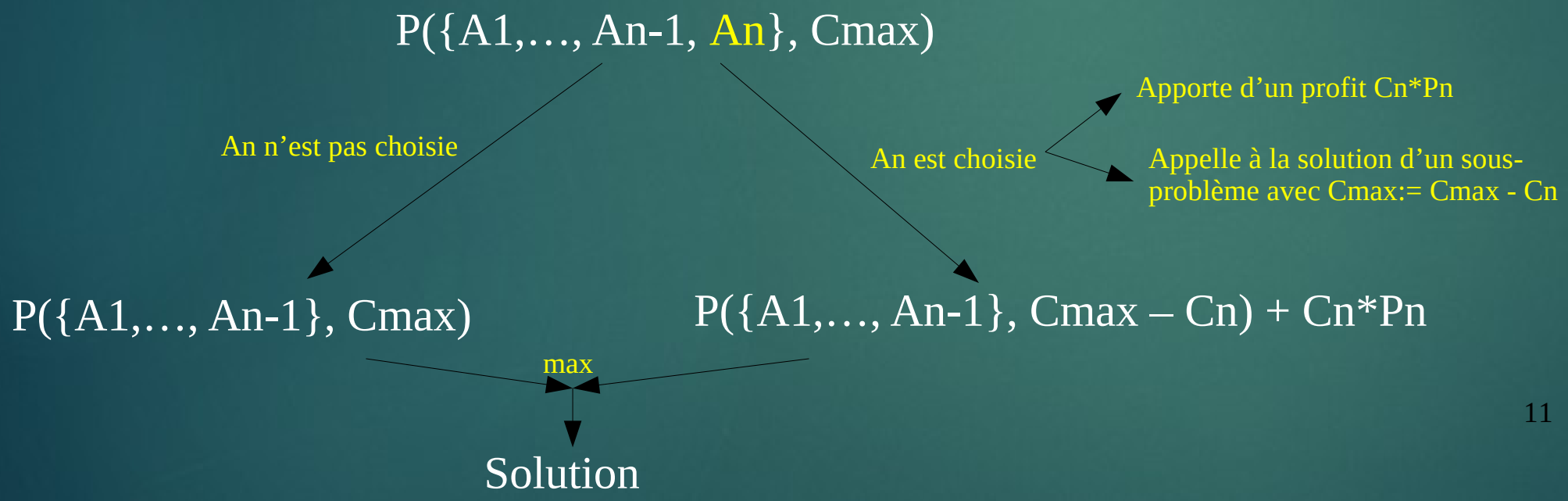
Algorithme Programmation dynamique

	1	...	...	C_{max}
...	...		...	...
An-1	...	P({A1, …,An-1}, C_{max} - Cn)	...	P({A1, …, An-1}, C_{max})
An	...	...	...	P({A1,…, An-1, An}, C_{max})



Algorithme Programmation dynamique

	1	...	...	C_{max}
...	...		...	...
An-1	...	P({A1, …,An-1}, C_{max} - Cn)	...	P({A1, …, An-1}, C_{max})
An	...	...	...	P({A1,…, An-1, An}, C_{max})



Programmation dynamique
Pseudocode

Complexité

Mémoire / Temporelle

*O(n*m)* *O(n*m)*

n = nombre_de_actions + 1
m = int(C_{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 opérateurs*
Pour j de 1 à m :
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 – coût(action-i)] + profit(action-i))
Sinon :
B[i][j] = B[i-1][j]

solution = [] *~ n éléments*
# tracer la solution *~ n*m opérateurs*
Tant que $C_{max} \geq 0$ et $n \geq 0$:
Si B[n][C_{max}] == B[n-1][C_{max} - coût(action-n)] + profit(action-n) :
l'action-n est choisie donc l'ajouter dans la solution
$C_{max} = C_{max}$ - coût(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 efficacités dans la limite du coût maximal autorisé.

Algorithme Greedy - Exemple

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

- Liste d'actions triées : [Action-2, Action-3, Action-1]

- Si C_{max} = 20, action 2 est choisie car C2 = 15 < 20
- Si C_{max} = 28, action 2 est choisie, puis action 3 car C2 + C3 = 15 + 5 < 28
- Si C_{max} = 31, action 2 est choisie, puis action 3, enfin action 1 car C2 + C3 + C1 = 15 + 5 + 10 < 31

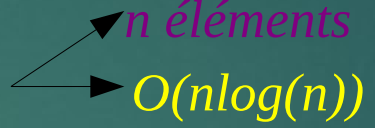
Greedy - Pseudocode

Complexité

Mémoire / Temporelle
O(n) *O(nlog(n))*

n = nombre_de_actions
L = liste des actions triées en ordre décroissant de l'efficacité
total_coût = 0
i = 1

solution = []
Tant que i ≤ n et total_coût ≤ C_{max} :
    action-i = L[i]
    total_coût = total_coût + coût(action-i)
    i = i+1
    ajouter action-i à la solution



=> n opérateurs => O(n)

Résumé

n : nombre d'actions
m : coût total maximal autorisé

	Complexité en mémoire	Complexité temporelle	Remarque
Brute Force	O(n)	$O(n*2^n)$	Temps de calcul exponentiel
Programmation dynamique	O(n*m)	O(n*m)	- Temps de calcul important si m est grand - Coût total autorisé et le coût de chaque action doivent être un nombre entier
Greedy	O(n)	O(nlog(n))	- Très rapide - Solution optimale locale