

# Evaluation des algorithmes de ranking pour classer des produits sur un site de e-commerce

Flavie Bertrand et Marion Tremblay

2025-12-08

## 1. Description du problème et objectif

### 1.1. Problème général de ranking

On considère :

- $i \in \{1, 2, \dots, n\}$  : l'élément  $i$
- un score associé à chaque élément :  $s_i \in \mathbb{R}$
- une variable binaire  $x_{i,p} \in \{0, 1\}$  qui vaut 1 si l'élément  $i$  est placé à la position  $p$  (0 sinon)
- un poids  $w_p$  associé à la position  $p$

L'objectif est de déterminer un classement des  $k$  premiers éléments maximisant le score total.

$$\max_x \sum_{i=1}^n \sum_{p=1}^k w_p s_i x_{i,p}$$

**Sous contraintes :**

- Un élément ne peut occuper au plus qu'une seule position :

$$\sum_{p=1}^k x_{i,p} \leq 1, \quad \forall i$$

- Chaque position doit être occupée par exactement un élément :

$$\sum_{i=1}^n x_{i,p} = 1, \quad \forall p$$

- Contraintes de groupes :

On définit des groupes :

$$G_g \subseteq \{1, \dots, m\}, \quad g = 1, \dots, G$$

On impose que chaque groupe  $G_g$  apparaisse au plus  $\alpha_g$  fois dans les  $k$  premiers :

$$\sum_{i \in G_g} \sum_{p=1}^k x_{i,p} \leq \alpha_g, \quad \forall g$$

## 1.2. Exemple concret

### 1.2.1. Contexte

On applique ce problème pour sélectionner les 10 produits les plus pertinents pour les afficher sur la page d'accueil d'un site de e-commerce. Chaque produit a un score  $s_i$  correspondant à la moyenne des notes données par les utilisateurs. On souhaite maximiser ce score tout en respectant certaines contraintes : - Diversité des catégories : pas plus de 3 produits de la même catégorie - Limiter la domination d'une marque : pas plus de 2 produits de la même marque - Contrainte marketing : au moins un produit sponsorisé doit apparaître dans le top 10

Pour répondre à ce problème on utilise la base de données ...

### 1.2.2. Formulation mathématique

La fonction objectif est :

$$\max \sum_{k=1}^{10} w_k s_{\pi(k)}$$

où :

- $\pi(k)$  est le produit placé à la position  $k$ ,
- $w_k$  est le poids associé à la position  $k$ .

#### Sous contraintes

- Chaque position contient exactement un produit :

$$\sum_{i=1}^n x_i^k = 1 \quad \forall k$$

- Un produit ne peut être affiché qu'une seule fois :

$$\sum_{k=1}^{10} x_i^k \leq 1 \quad \forall i$$

- Maximum 3 produits par catégorie :

$$\sum_{i \in \text{cat}} \sum_{k=1}^{10} x_i^k \leq 3$$

- Maximum 2 produits par marque :

$$\sum_{i \in \text{brand}} \sum_{k=1}^{10} x_i^k \leq 2$$

- Au moins 1 produit sponsorisé dans le top 10 :

$$\sum_{i \in \text{sponsor}} \sum_{k=1}^{10} x_i^k \geq 1$$

Cette dernière contrainte est équivalente à avoir maximum 9 produits non sponsorisé dans le top 10, soit :

$$\sum_{i \in \text{no sponsor}} \sum_{k=1}^{10} x_i^k \leq 9$$

## 2. Approche heuristique

L'approche **gloutonne** constitue une première stratégie simple pour résoudre le problème de ranking pondéré avec contraintes de groupes.

### Principe algorithmique

1. Initialiser les compteurs de chaque groupe à 0 et liste des éléments sélectionnés vide.
2. Pour chaque position ( $p = 1, \dots, k$ ) :
  - Considérer les éléments non encore sélectionnés respectant les contraintes de groupes.
  - Choisir l'élément avec le gain immédiat maximal  $w_p \cdot s_i$
  - Mettre à jour les compteurs de groupes et le score total.
3. Arrêt si aucune sélection possible pour une position ou si toutes les positions sont remplies.

Cette méthode est **très rapide** et facile à implémenter, mais elle ne garantit **pas d'optimalité globale**, car elle ne considère pas les effets futurs de chaque choix.

### Exemple illustratif

##	id	score	groupes	position	poids_position	score_pondere
## 1	10	10	A	1	5	50
## 2	2	9	A	2	4	36
## 3	3	8	B	3	3	24
## 4	4	7	B	4	2	14
## 5	5	6	C	5	1	6

- L'algorithme privilégie **les éléments les mieux notés pour les premières positions**.
- Les contraintes de groupe peuvent limiter le choix, ce qui peut conduire à **des choix sous-optimaux pour les positions suivantes**.
- Malgré ses limites, cette approche **rapide et simple** est utile pour obtenir une **baseline** ou comparer avec des méthodes plus sophistiquées comme la programmation dynamique exacte ou le Beam Search.

### 3. Solution 1 : programmation dynamique

#### Idée générale

Pour résoudre le problème de ranking sous contraintes de groupe et de poids positionnels, on utilise une programmation dynamique :

Chaque état est défini par : - le nombre de positions déjà remplies, - le nombre d'éléments choisis par groupe (compteurs de groupes). - À chaque étape (position p), on essaie tous les éléments possibles qui respectent les contraintes de groupes et qui n'ont pas déjà été choisis.

Pour chaque état, on garde le meilleur score pondéré atteint.

À la fin, on récupère l'état final avec le score total maximal et la sélection correspondante.

#### Exemple concret

On applique l'algorithme sur un petit jeu de données illustratif : - 10 produits avec un score utilisateur (score) et un ou plusieurs groupes (groupes). - Contraintes : max 2 produits du groupe A, 2 du groupe B, 1 du groupe C. - Poids décroissants pour les positions : 5, 4, 3, 2, 1.

Table 1: Tableau des éléments sélectionnés par l'algorithme ranking max

	Position	ID	Score	Groupe	Poids position	Score pondéré
<b>10</b>	1	10	10	A	<b>5</b>	50
<b>2</b>	2	2	9	A	<b>4</b>	36
<b>3</b>	3	3	8	B	<b>3</b>	24
<b>4</b>	4	4	7	B	<b>2</b>	14
<b>5</b>	5	5	6	C	<b>1</b>	6

Score total = 130

Visualisation : contribution par position



Le graphique montre que les positions les mieux pondérées (1 et 2) contribuent le plus au score total. Il permet de visualiser l'impact de chaque choix sur le score global.

## 4. Solution 2 : programmation dynamique et tas

Pour des ensembles de données plus volumineux, la **programmation dynamique exacte** peut devenir très coûteuse en temps et en mémoire, car le nombre d'états possibles croît de manière exponentielle avec le nombre de positions ( $k$ ) et le nombre de groupes ( $G$ ).

Pour pallier cette limitation, nous utilisons une approche **approximate avec Beam Search**, elle conserve uniquement `beam_size` meilleurs états à chaque niveau, on a alors une approximation proche de l'optimum.

### Principe algorithmique

1. Représentation des états, chaque état est défini par :
  - le score cumulé des éléments sélectionnés,
  - le compteur d'éléments choisis par groupe,
  - une clé unique codant ces compteurs.
2. Trier les éléments par score décroissant.
3. Pour chaque position :
  - Pour chaque état conservé, on génère de nouveaux états en ajoutant chaque élément disponible respectant les contraintes.
  - On ajoute ces états dans un max-heap et ne garder que `beam_size` meilleurs états (pruning).
4. À la fin, on reconstruit la sélection à partir du meilleur état final.

La largeur du beam (`beam_size`) permet de **contrôler le compromis précision / performance**.

## Un exemple

On reprend l'exemple précédent des 10 produits, avec un **beam de taille 5** :

```
## $best_score
## [1] 40
##
## $selected_items
## [1] 2 3 4 5 8
##
## $beam_size
## [1] 5
##
## $approximation
## [1] TRUE
```

- `res$best_score` donne le score cumulé approximatif.
- `res$selected_items` contient la liste des éléments sélectionnés.
- Le champ `approximation` indique si la solution a été tronquée par le beam.
- Même avec un **beam très limité**, l'algorithme tend à **sélectionner les éléments avec les meilleurs scores** tout en respectant les contraintes de groupe.
- La **programmation dynamique combinée à un heap** permet de **garder les meilleures solutions intermédiaires** sans générer tous les états possibles, ce qui est crucial pour de grands ensembles de données.
- Ce compromis entre **exactitude et performance** est particulièrement utile dans les systèmes de recommandation ou e-commerce où le calcul doit rester rapide.

## 5. Comparaison des résultats

## 6. Complexité des algorithmes (par le calcul)

Voici nos 4 algorithmes :

1. Algorithme naïf R
2. Algorithme dynamique R
3. Algorithme dynamique C++
4. Algorithme dynamique C++ amélioré (beam search)

Pour chacun, nous analysons la complexité **temps** (meilleur, pire, moyenne) Les paramètres en jeu sont :  
- **n** : nombre d'items (taille de l'entrée) - **G** : nombre de groupes - **k** : nombre de positions à remplir - **S** : nombre d'états effectivement atteints dans la DP - **M** : nombre théorique maximal d'états =  $\prod_{g=1}^G (\max\_cap_g + 1)$   
-  $\bar{g}$  : nombre moyen de groupes par item

### 1. Algorithme Naïf Glouton (R)

Sélection gloutonne : pour chaque position  $p$ , parcourir tous les candidats restants et choisir celui au meilleur gain immédiat  $w_p \times score_i$ .

## Analyse détaillée

### Étape 1 - Initialisation : $O(G)$

- Création des structures de données
- Négligeable devant la boucle principale

### Étape 2 - Boucle principale

Pour chaque position  $p = 1, \dots, k$  :

- Nombre d'items restants à examiner :  $n - (p - 1)$
- Pour chaque item candidat :
  - Parser les groupes :  $O(\bar{g})$
  - Vérifier les contraintes :  $O(\bar{g})$  comparaisons
  - Calculer le gain :  $O(1)$
- Mise à jour des compteurs :  $O(\bar{g})$

Coût pour la position  $p$  :

$$T_p = O((n - p + 1) \times \bar{g})$$

Coût total de la boucle :

$$T_{\text{boucle}} = \sum_{p=1}^k O((n - p + 1) \times \bar{g}) = O\left(\bar{g} \times \left(kn - \frac{k(k-1)}{2}\right)\right) \approx O(kn\bar{g})$$

## Complexité temporelle

$$T_{\text{naïf}}(n) = O(kn\bar{g})$$

**Meilleur cas** :  $T(n) = O(kn)$  si  $\bar{g} = O(1)$

**Cas moyen** :  $T(n) = O(kn\bar{g})$

**Pire cas** :  $T(n) = O(knG)$  si  $\bar{g} = G$

## 2. Algorithme DP (R)

Programmation dynamique avec états définis par  $(p, c_1, \dots, c_G)$  où  $c_g$  = nombre d'items du groupe  $g$  déjà utilisés.

## Analyse détaillée

### Étape 1 - Prétraitement : $O(nG)$

- Parser les groupes de chaque item :  $O(n\bar{g})$
- Créer la matrice binaire `item_groups` :  $O(nG)$

## Étape 2 - Programmation dynamique

Structure :  $\text{DP}[[p+1]][[\text{key}]]$  avec  $\text{key} = "c1, c2, \dots, cG"$

Pour chaque position  $p = 1, \dots, k$  :

- États actifs au niveau  $p - 1$  :  $S_p$  états
- Pour chaque état actif :
  - Parser la clé :  $O(G)$
  - Pour chaque item  $i = 1, \dots, n$  :
    - \* Vérifier si utilisé :  $O(p)$
    - \* Calculer nouveaux compteurs :  $O(G)$
    - \* Vérifier contraintes :  $O(G)$
    - \* Créer nouvelle clé :  $O(G)$
    - \* Mise à jour DP :  $O(1)$

Coût pour un état et un item :  $O(p + G)$

Coût pour la position  $p$  :  $T_p = O(S_p \times n \times (p + G))$

En supposant  $S_p \approx S$  constant et  $G \geq k$  :

$$T_{\text{DP}} = \sum_{p=1}^k O(S \times n \times G) = O(SnkG)$$

**Étape 3 - Recherche du meilleur** :  $O(S \times G)$

## Complexité temporelle

$$\boxed{T_{\text{DP-R}}(n) = O(SnkG)}$$

**Meilleur cas** :  $\boxed{T(n) = O(nkG)}$  si  $S = O(1)$

**Cas moyen** :  $\boxed{T(n) = O(SnkG)}$  avec  $S \ll M$

**Pire cas** :  $\boxed{T(n) = O(MnkG)}$  où  $M = \prod_{g=1}^G (\text{max\_cap}_g + 1)$  (exponentiel en  $G$ )

## 3. Algorithme DP (C++)

Même logique que DP R, mais avec `unordered_map` et optimisations C++.

### Analyse détaillée

**Prétraitement** :  $O(nG)$

- Parsing avec `stringstream` :  $O(n\bar{g})$
- Construction matrice :  $O(nG)$

### Programmation dynamique

Pour chaque position  $p$ , chaque état  $S_p$ , chaque item  $n$  :



- Parse key :  $O(G)$
- Vérifier si utilisé (std::find) :  $O(p)$
- Calculer compteurs :  $O(G)$
- Make key :  $O(G)$
- Hash + lookup :  $O(G)$  pour le hashing,  $O(1)$  amorti pour l'accès
- Insert/update :  $O(1)$  amorti

Coût par transition :  $O(p + G)$

Si  $G \geq k$  :  $T_{DP} = O(SnkG)$

### Complexité temporelle

$$T_{DP-C++}(n) = O(SnkG)$$

**Meilleur cas** :  $T(n) = O(nkG)$  si  $S = O(1)$

**Cas moyen** :  $T(n) = O(SnkG)$

**Pire cas** :  $T(n) = O(MnkG)$  avec  $M$  exponentiel en  $G$

**Note** : Même complexité asymptotique que DP R, mais facteur constant 5-20× plus petit en pratique.

## 4. Algorithme dynamique amélioré (C++)

DP avec pruning : conserve uniquement les `beam_size` meilleurs états à chaque niveau.

### Analyse détaillée

Soit  $B = \min(\text{beam\_size}, M)$  le nombre effectif d'états conservés.

**Étape 1 - Tri initial** :  $O(n \log n)$

Items triés par score décroissant pour améliorer la qualité des états gardés.

**Étape 2 - DP avec tas (priority\_queue)**

Pour chaque position  $p$ , chaque état ( $\max B$ ), chaque item  $n$  :

- Extraction des états du tas :  $O(B \log B)$  (une fois par niveau)
- Pour chaque état  $\times$  item :
  - Calcul compteurs :  $O(G)$
  - Make key :  $O(G)$
  - Push dans tas :  $O(\log B)$
  - Pruning si nécessaire :  $O(\log B)$

Coût par niveau :  $T_p = O(B \log B) + O(nB(G + \log B))$

Si  $G \geq \log B$  (généralement vrai) :

$$T_p = O(nBG)$$

Sur  $k$  niveaux :

$$T_{\text{DP}} = O(nkBG)$$

**Étape 3 - Recherche meilleur** :  $O(kB)$

**Étape 4 - Reconstruction** :  $O(n)$

**Complexité temporelle**

$$T_{\text{Beam}}(n) = O(n \log n + nkBG)$$

Si  $nkBG \gg n \log n$  :  $T_{\text{Beam}}(n) = O(nkBG)$

**Meilleur cas** :  $T(n) = O(n \log n + nkG)$  si  $B = 1$

**Cas moyen** :  $T(n) = O(n \log n + nkBG)$  avec  $B$  modéré (100-10000)

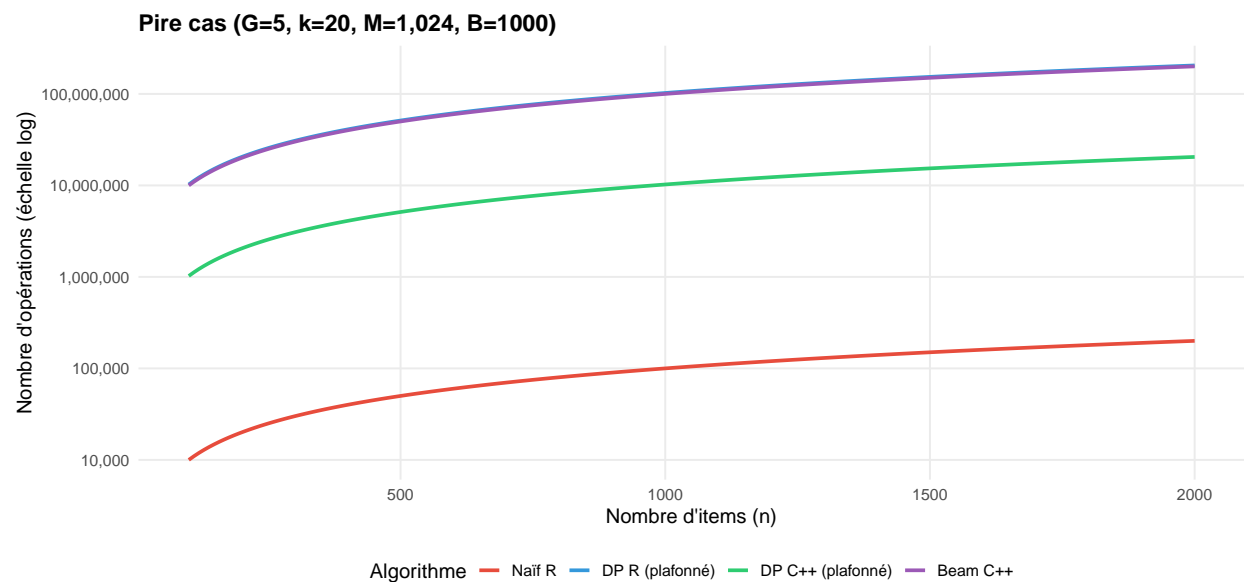
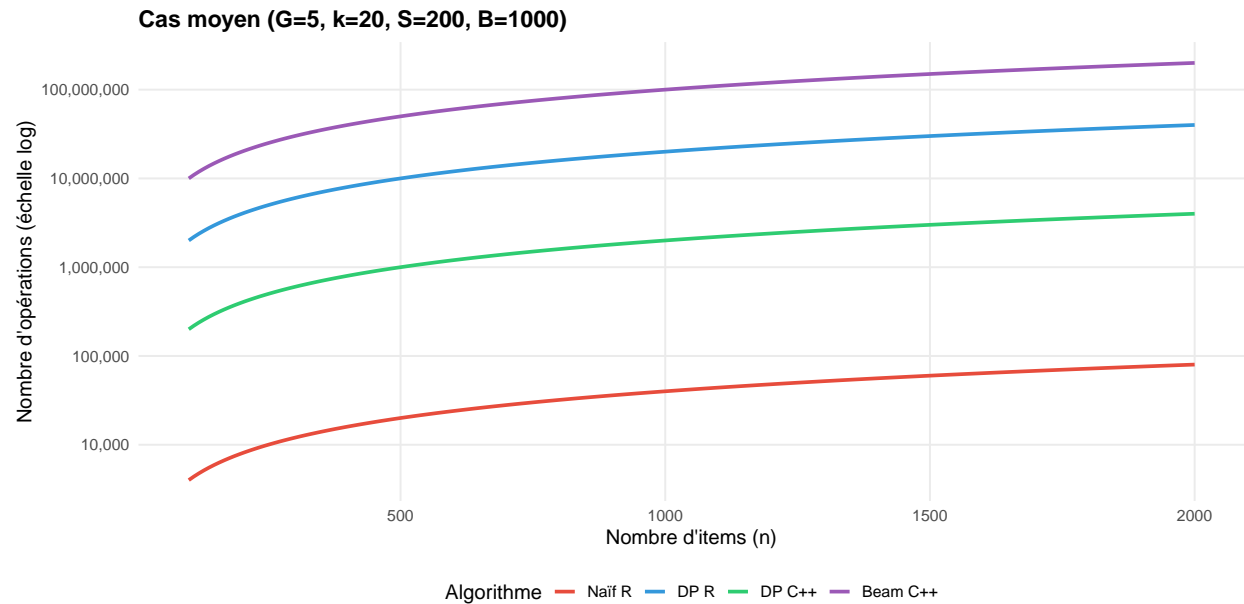
**Pire cas** :  $T(n) = O(n \log n + nkBG)$  (même formule,  $B$  peut être grand)

**Avantage majeur** : Complexité **contrôlable** via `beam_size`, contrairement aux versions exactes.

## Tableau récapitulatif

Algorithme	Meilleur cas	Cas moyen	Pire cas
<b>Naïf R</b>	$O(kn)$	$O(kn\bar{g})$	$O(knG)$
<b>DP R</b>	$O(nkG)$	$O(SnkG)$	$O(MnkG)$
<b>DP C++</b>	$O(nkG)$	$O(SnkG)$	$O(MnkG)$
<b>Beam C++</b>	$O(n \log n + nkG)$	$O(n \log n + nkBG)$	$O(n \log n + nkBG)$

## Comparaison graphique



Note: DP R et DP C++ plafonnés à  $10^{12}$  pour la lisibilité

## Observations

### Cas moyen :

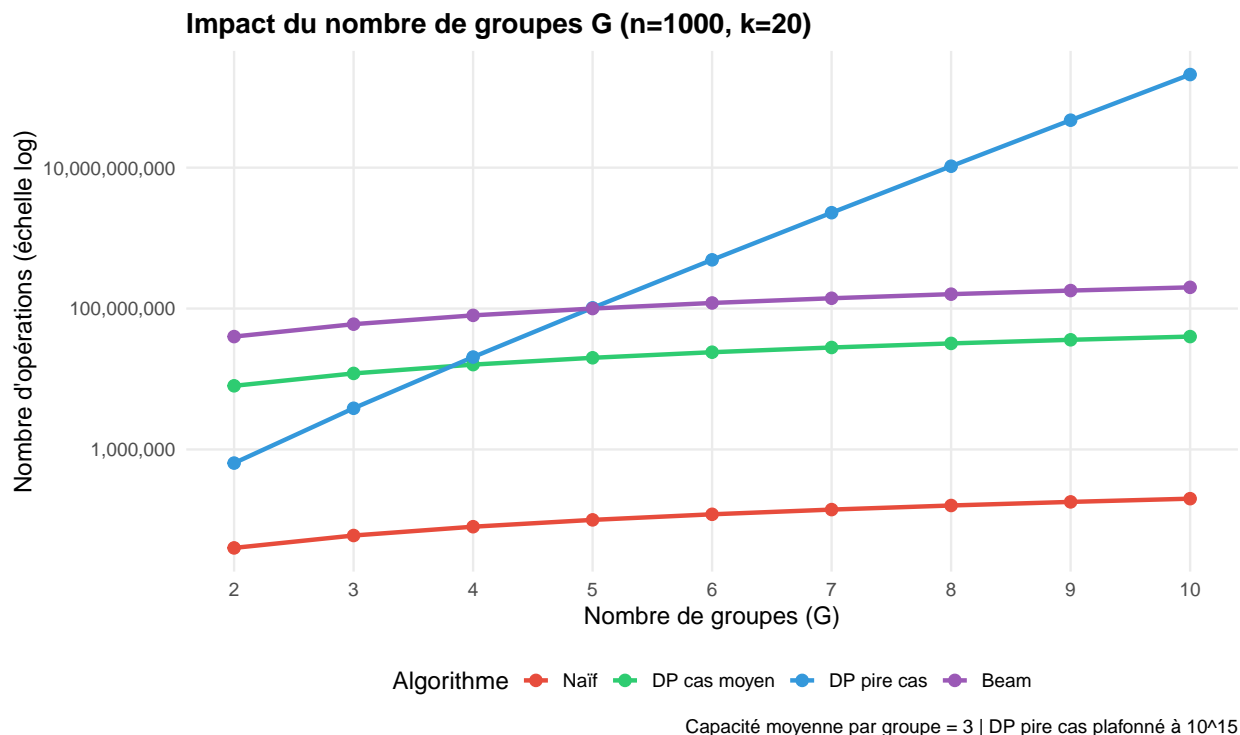
- Beam Search et DP C++ sont très compétitifs
- DP R significativement plus lent (facteur 10)
- Naïf acceptable pour petites instances

### Pire cas :

- Explosion exponentielle des DP exact (R et C++)

- Beam Search reste linéaire et prévisible
- Seul algorithme viable pour grands problèmes avec  $G \geq 6$

## Impact du nombre de groupes G



## Analyse

- $G \leq 4$  : Tous les algorithmes sont viables
- $G = 5-6$  : DP commence à être problématique en pire cas
- $G \geq 7$  : Seul Beam Search reste praticable
- L'explosion de  $M = (cap+1)^G$  rend DP exact inutilisable pour grand G

## Choix de l'algorithme selon le contexte

Contexte	Algorithme recommandé	Justification
Petits problèmes ( $n < 100$ , $G < 3$ , $k < 10$ )	<b>Naïf R</b> ou <b>DP R</b>	Simple, rapide, optimal
Moyens problèmes ( $n < 1000$ , $G < 5$ , $k < 20$ )	<b>DP C++</b>	Optimal garanti, performance acceptable
Grands problèmes ( $n > 1000$ ou $G > 5$ )	<b>Beam C++</b>	Seul viable, quasi-optimal
Production avec contraintes temps	<b>Beam C++</b> (B=5000)	Prédictible, contrôlable
Recherche d'optimum prouvé	<b>DP C++</b> si faisable	Exact mais peut échouer

$$\text{Si } M = \prod_{g=1}^G (\text{max\_cap}_g + 1) > 100000 \Rightarrow \text{Utiliser Beam Search}$$

## 7. Temps de calcul