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 :

- $i \in \{1, 2, \ldots, n\}$: 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 :

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

- Contraintes de groupes :

On définit des groupes :

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

On impose que chaque groupe G_g apparaisse au plus α_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 5 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 2 produits de la même catégorie - Contrainte marketing : au moins un produit sponsorisé doit apparaître dans le top 10

Pour répondre à ce problème on utilise la base de données des retours produits du site Amazon datant de 2018. Nous avons sélectionné un échantillon de 250 produits de cette base de données. Nous avons généré une colonne en plus pour réaliser la contrainte marketing, en associant 15% des produits de la base de données à des produits sponsorisés.

1.2.2. Formulation mathématique

La fonction objectif est :

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

où :

- $x_{i,p}$ est une variable binaire qui vaut 1 si l'élément i est placé à la position p (0 sinon)
- w_p est le poids associé à la position p: il vaut 5 à la position puis est décrémenter de 1 à chaque position suivante.

Sous contraintes

- Chaque position contient exactement un produit :

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

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

$$\sum_{k=1}^{5} x_{i,k} \leq 1 \quad \forall i$$

- Maximum 2 produits par catégorie :

$$\sum_{i \in \text{cat } c} \sum_{p=1}^{5} x_{i^p} \leq 2$$

- Au moins 1 produit sponsorisé dans le top 5 :

$$\sum_{i \in \text{sponsor}} \sum_{p=1}^{5} x_{i,p} \geq 1$$

Cette dernière contrainte est équivalente à avoir maximum 4 produits non sponsorisé dans le top 5, soit :

$$\sum_{i \in \text{no sponsor}} \sum_{p=1}^{5} x_{i,p} \leq 4$$

2. Approche heuristique

L'approche **heuristique** 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,. . . ,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

```
## [1] 71.8
```

```
##   id score                  groupes position poids_position score_pondere
## 1  8   4.8     Beauty,Not_sponsored        1              5          24.0
## 2 11   4.8     Office,Not_sponsored        2              4          19.2
## 3 16   4.8     Office,Not_sponsored        3              3          14.4
## 4 19   4.8 Toys_Games,Not_sponsored        4              2           9.6
## 5 32   4.6          Books,Sponsored        5              1           4.6
```

```
## [1] 71.8
```

- 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.
- Cette approche est rapide et simple.

3. Solution 1 : programmation dynamique

L'algorithme

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.

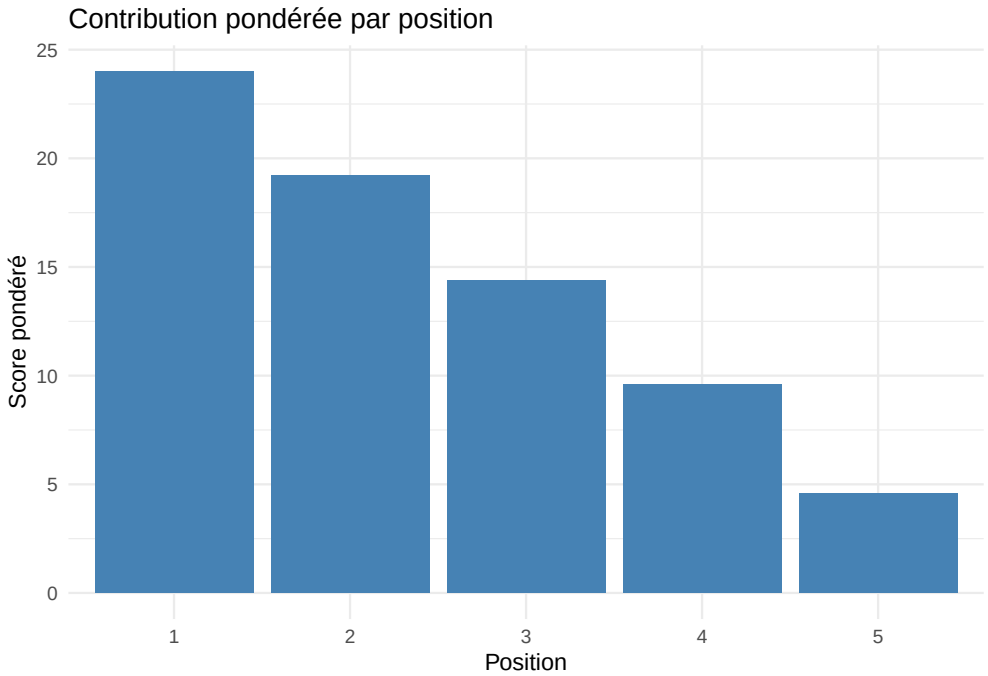
Exemple concret

On applique l'algorithme sur le jeu de données décrit en introduction :

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

	Position	ID	Score	Groupe	Poids position	Score pondéré
8	1	8	4.8	Beauty , Not_sponsored	**5**	24.0
19	2	19	4.8	Toys_Games , Not_sponsored	**4**	19.2
11	3	11	4.8	Office , Not_sponsored	**3**	14.4
16	4	16	4.8	Office , Not_sponsored	**2**	9.6
36	5	36	4.6	Toys_Games, Sponsored	**1**	4.6

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 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 améliorer l'algorithme, 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.

Un exemple

On reprend l'exemple précédent avec un **beam de taille 5** :

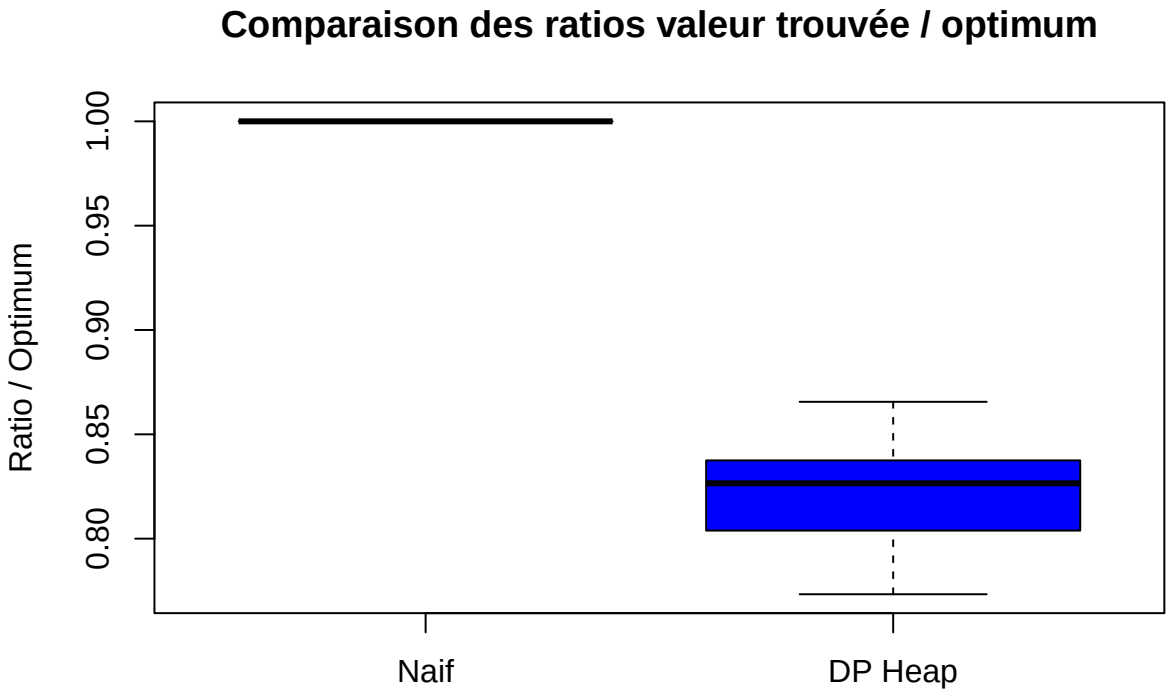
```
## $selected_items
##   index score                    groupes position poids_position score_pondere
## 1    26   3.4       Beauty,Not_sponsored        1              1           3.4
## 2    50   3.9 Home_Kitchen,Not_sponsored        2              1           3.9
## 3     7   4.0       Beauty,Not_sponsored        3              1           4.0
## 4    48   4.0      Electronics,Sponsored        4              1           4.0
## 5    20   4.1 Home_Kitchen,Not_sponsored        5              1           4.1
##
## $best_score
## [1] 19.4
##
## $beam_size
## [1] 5
##
## $approximation
## [1] TRUE
```

- 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.

5. Comparaison de la performance

```
set.seed(13)

num_instances <- 50   # nombre d'instances
n_items <- 50         # items par instance
k <- 5                # positions
ratios1 <- numeric(num_instances)
ratios2 <- numeric(num_instances)


df <- read.csv("amazon_products_250.csv", stringsAsFactors = FALSE)
df$sponsored <- ifelse(df$sponsored == "True", "Sponsored", "Not_sponsored")

# Poids de position
poids_positions <- seq(k, 1, by = -1)

# Création des listes de contraintes
liste_cat <- setNames(as.list(rep(2, length(unique(df$category)))),
                      unique(df$category))
max_par_groupe <- c(liste_cat, list(Not_sponsored = 9))

for (i in 1:num_instances)
{
  echantillon <- df[sample(nrow(df), min(n_items, nrow(df))), ]
  data_mat <- data.frame(
    id = 1:nrow(echantillon),
    score = echantillon$score,
    groupes = paste(echantillon$category,
                    echantillon$sponsored, sep = ",")
  )
  V_g <- ranking_naif_max(data_mat,k,max_par_groupe,poids_positions)$best_score
  V_opt <- ranking_max_cpp(data_mat,k,max_par_groupe,poids_positions)$best_score
  V_dp_heap <- ranking_max_dp_heap_cpp(data_mat,k,max_par_groupe,poids_positions)$best_score

  ratios1[i] <- V_g / V_opt
  ratios2[i] <- V_dp_heap / V_opt

}
```

```
boxplot(ratios1, ratios2, names=c("Naif", "DP Heap"),
        main="Comparaison des ratios valeur trouvée / optimum",
        ylab="Ratio / Optimum", col=c("red","blue"))
```



On constate que l'approche heuristique fonctionne très bien alors que la programmation dynamique avec tas ne trouve pas toujours la solution optimale.

Si l'algorithme naif trouve presque toujours la solution optimale, voici un exemple où ce n'est pas le cas :

```
data <- data.frame(
  id = 1:6,
  score = c(100, 99, 98, 97, 60, 50),
  groupes = c("A,S", "A,S", "A", "B,S", "C,S", "C,S")
)
max_par_groupe <- list(A = 2, B = 2, C = 2, S = 4)
poids_positions <- seq(k, 1, by = -1)

cat("Resultat naif\n")
```

```
## Resultat naif
```

```
resultat_naif <- ranking_naif_max(data, 5, max_par_groupe, poids_positions)
```

```
## Warning in ranking_naif_max(data, 5, max_par_groupe, poids_positions):
## Impossible de remplir toutes les 5 positions. Seulement 4 positions remplies.
```

```
print(resultat_naif)
```

```
## $selected_items
```

```
##   id score groupes position poids_position score_pondere
## 1  1   100     A,S        1              5           500
## 2  2    99     A,S        2              4           396
## 3  4    97     B,S        3              3           291
## 4  5    60     C,S        4              2           120
##
## $best_score
## [1] 1307
```

```
cat("Resultat programmation dynamique\n")
```

```
## Resultat programmation dynamique
```

```
resultat_dp <- ranking_max_cpp(data, 5, max_par_groupe, poids_positions)
print(resultat_dp)
```

```
## $selected_items
##   index score groupes position poids_position score_pondere
## 1     1   100     A,S        1              5           500
## 2     3    98       A        2              4           392
## 3     4    97     B,S        3              3           291
## 4     5    60     C,S        4              2           120
## 5     6    50     C,S        5              1            50
##
## $best_score
## [1] 1353
```

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 = $\prod_{g=1}^{G}(\text{max\_cap}_g + 1)$
- $\bar{g}$: 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 $w_p \times \text{score}_i$.

Analyse détaillée

Étape 1 - Initialisation : $O(G)$

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

Étape 2 - Boucle principale

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

- 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

$$\boxed{T_{\text{naïf}}(n) = O(kn\bar{g})}$$

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

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

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

2. Algorithme DP (R)

Programmation dynamique avec états définis par $(p, c_1, \ldots, c_G)$ 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 : `DP[[p+1]][[key]]` avec `key = "c1,c2,...,cG"`

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

- États actifs au niveau $p-1$: S_p états
- Pour chaque état actif :
  - Parser la clé : $O(G)$
  - Pour chaque item $i = 1, \ldots, n$:
    - 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 $G \geq k$: $T_{\text{DP}} = O(SnkG)$

Complexité temporelle

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

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

Cas moyen : $\boxed{T(n) = O(SnkG)}$

Pire cas : $\boxed{T(n) = O(MnkG)}$ 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 × 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

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

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

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

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

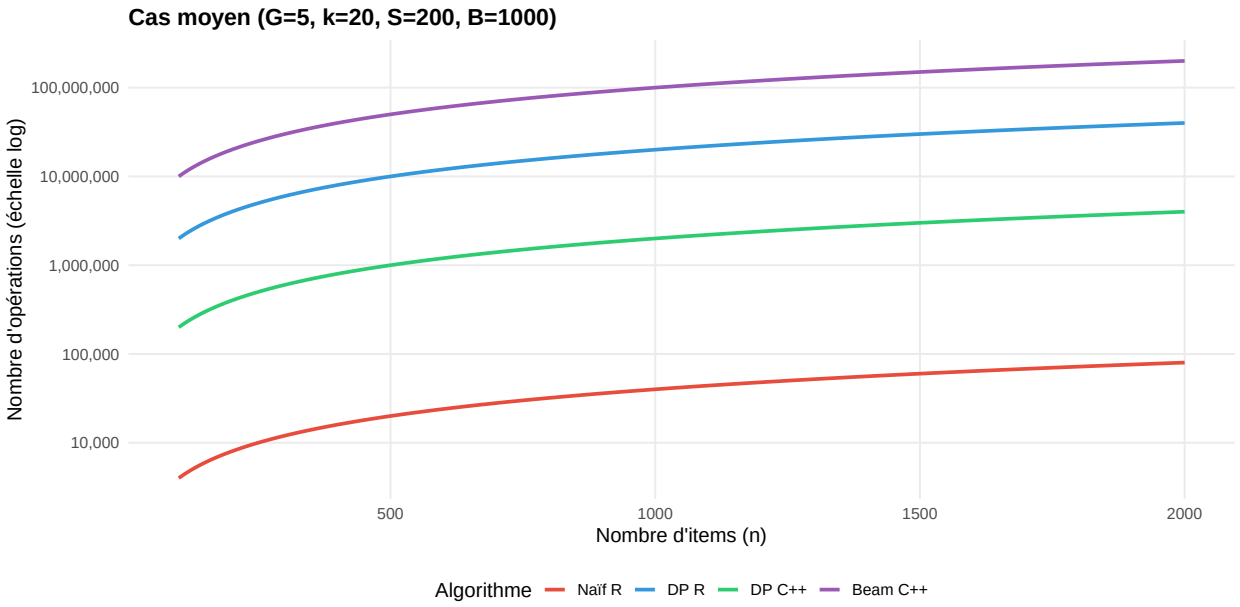
Pire cas : $\boxed{T(n) = O(n \log n + nkBG)}$ (même formule, B peut être grand)

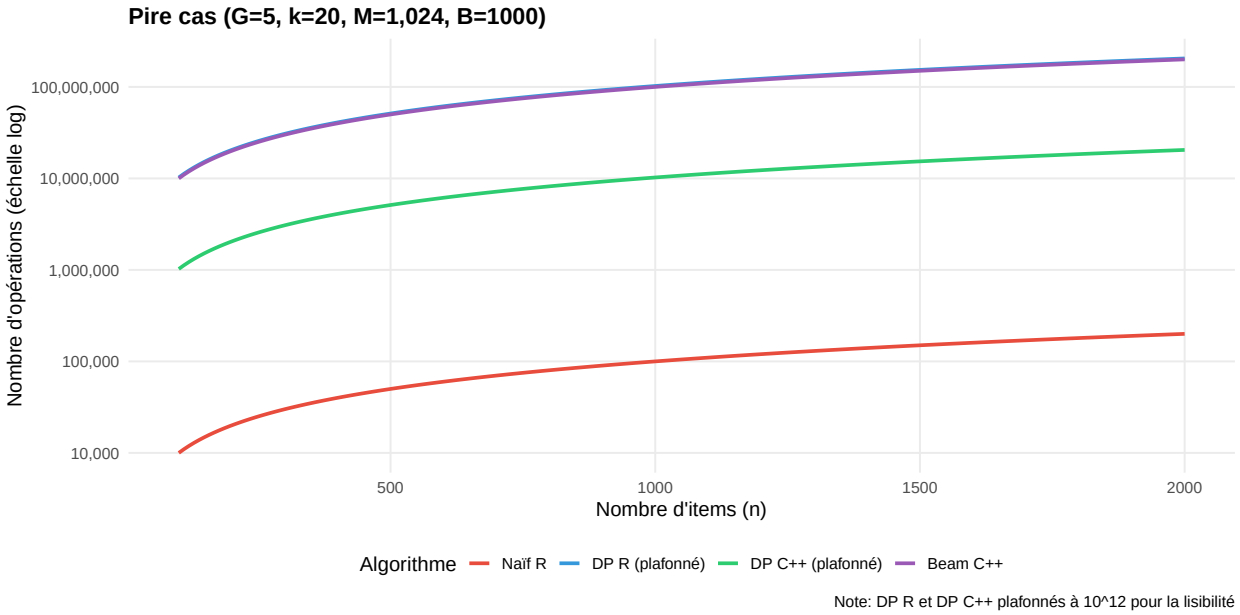
Avantage majeur : Complexité **contrôlable** via `beam_size`, contrairement aux versions exactes.

Tableau récapitulatif

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

Comparaison graphique





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

Observations

Cas moyen :

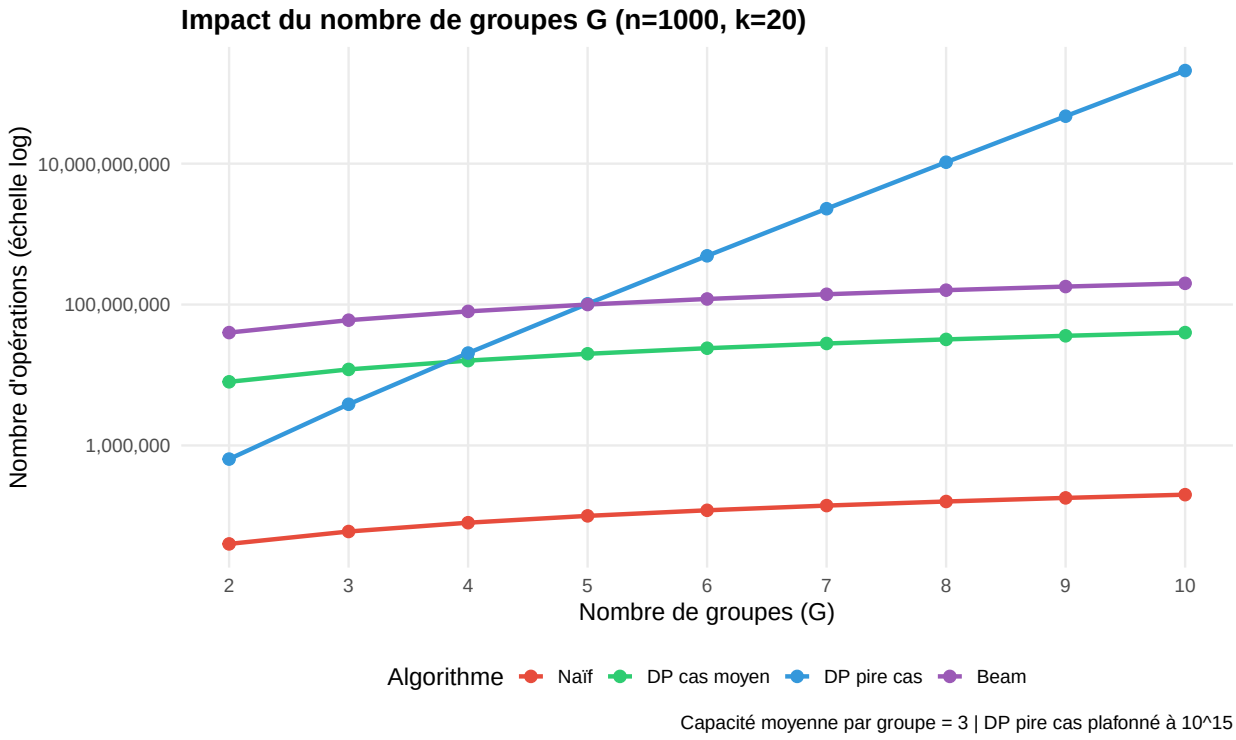
- Beam Search et DP C++ sont très compétitifs
- DP R significativement plus lent (facteur 10)
- Naïf acceptable pour petites instances

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

7. Temps de calcul

```r
n <- c(20,22,25,32,42,55,70,90,125,160)
temps_moyens1 <- numeric(length(n))
temps_moyens2 <- numeric(length(n))
temps_moyens3 <- numeric(length(n))
temps_moyens4 <- numeric(length(n))
k <- 5

df <- read.csv("amazon_products_250.csv", stringsAsFactors = FALSE)
df$sponsored <- ifelse(df$sponsored == "True", "Sponsored", "Not_sponsored")

# Poids de position
poids_positions <- seq(k, 1, by = -1)

# Création des listes de contraintes
liste_cat <- setNames(as.list(rep(2, length(unique(df$category)))),
                      unique(df$category))
max_par_groupe <- c(liste_cat, list(Not_sponsored = 4))

for (i in 1:length(n))
{
  echantillon <- df[sample(nrow(df), min(n[i], nrow(df))), ]
  data_mat <- data.frame(
    id = 1:nrow(echantillon),
    score = echantillon$score,
    groupes = paste(echantillon$category,
                    echantillon$sponsored, sep = ",")
  )

  start_time1 <- Sys.time()
  res1 <- ranking_naif_max(data_mat, k = k, max_par_groupe, poids_positions = poids_positions)
  end_time1 <- Sys.time()
  elapsed_time1 <- as.numeric(difftime(end_time1, start_time1, units = "secs"))
  temps_moyens1[i] <- elapsed_time1

  start_time2 <- Sys.time()
  res2 <- ranking_max(data_mat, k = k, max_par_groupe, poids_positions = poids_positions)
  end_time2 <- Sys.time()
  elapsed_time2 <- as.numeric(difftime(end_time2, start_time2, units = "secs"))
  temps_moyens2[i] <- elapsed_time2

  start_time3 <- Sys.time()
  res3 <- ranking_max_cpp(data_mat, k = k, max_par_groupe, poids_positions = poids_positions)
  end_time3 <- Sys.time()
  elapsed_time3 <- as.numeric(difftime(end_time3, start_time3, units = "secs"))
  temps_moyens3[i] <- elapsed_time3

  start_time4 <- Sys.time()
  res4 <- ranking_max_dp_heap_cpp(data_mat, k = k, max_par_groupe, poids_positions = poids_positions)
  end_time4 <- Sys.time()
  elapsed_time4 <- as.numeric(difftime(end_time4, start_time4, units = "secs"))
  temps_moyens4[i] <- elapsed_time4
```

```
}

cat("Temps pour l'algorithme naif\n")
```

```
## Temps pour l'algorithme naif
```

```
temps_moyens1
```

```
##  [1] 0.01113391 0.01179814 0.01621509 0.01668715 0.02391195 0.03165507
##  [7] 0.03706503 0.04764700 0.07077289 0.08631182
```

```
cat("Temps pour l'algorithme de programmation dynamique\n")
```

```
## Temps pour l'algorithme de programmation dynamique
```

```
temps_moyens2
```

```
##  [1]  0.1158762  0.2700751  0.4668369  0.5885761  1.6892302  2.6166260
##  [7]  3.5821888  7.5632188 12.7354519 30.7873700
```

```
cat("Temps pour l'algorithme de programmation dynamique (en C++)\n")
```

```
## Temps pour l'algorithme de programmation dynamique (en C++)
```

```
temps_moyens3
```

```
##  [1] 0.003202200 0.005258083 0.008975029 0.009464025 0.021492958 0.030364990
##  [7] 0.038010120 0.064638138 0.091548920 0.196481943
```

```
cat("Temps pour l'algorithme de programmation dynamique avec tas (en C++)\n")
```

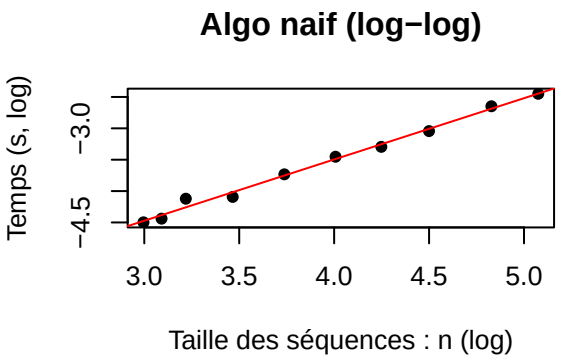
```
## Temps pour l'algorithme de programmation dynamique avec tas (en C++)
```

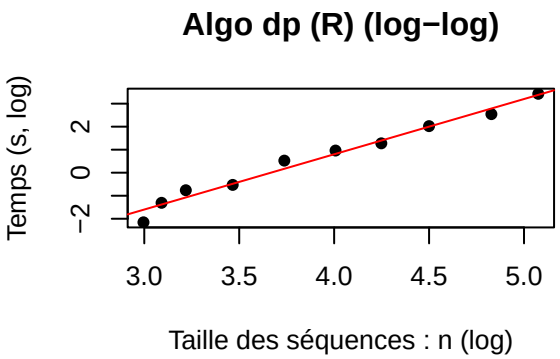
```
temps_moyens4
```

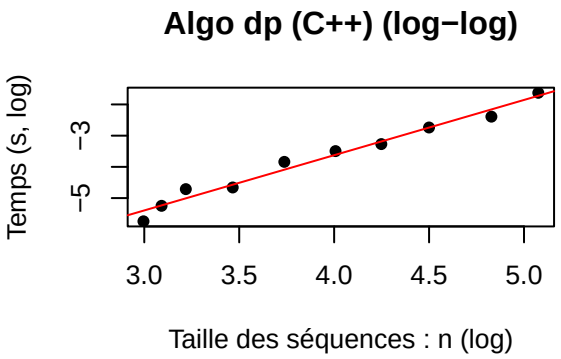
```
##  [1] 0.06514001 0.07589602 0.08943295 0.15454412 0.22930884 0.25486898
##  [7] 0.49783587 0.66043091 0.79649591 1.51366687
```

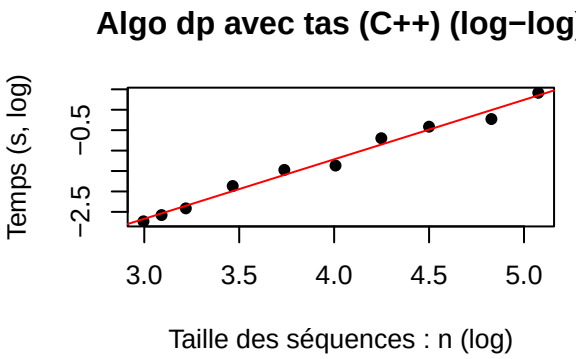
```
par(mfrow=c(2,2))

# Graphique 1
res1 <- data.frame(
  n = n,
  time = temps_moyens1
)
plot(log(res1$n), log(res1$time), pch=16,
     xlab="Taille des séquences : n (log)",
```

```
     ylab="Temps (s, log)",
     main="Algo naif (log-log)")
abline(lm(log(time) ~ log(n), data=res1), col="red")

# Graphique 2
res2 <- data.frame(
  n = n,
  time = temps_moyens2
)
plot(log(res2$n), log(res2$time), pch=16,
     xlab="Taille des séquences : n (log)",
     ylab="Temps (s, log)",
     main="Algo dp (R) (log-log)")
abline(lm(log(time) ~ log(n), data=res2), col="red")

# Graphique 3
res3 <- data.frame(
  n = n,
  time = temps_moyens3
)
plot(log(res3$n), log(res3$time), pch=16,
     xlab="Taille des séquences : n (log)",
     ylab="Temps (s, log)",
     main="Algo dp (C++) (log-log)")
abline(lm(log(time) ~ log(n), data=res3), col="red")

# Graphique 4
res4 <- data.frame(
  n = n,
  time = temps_moyens4
)
plot(log(res4$n), log(res4$time), pch=16,
     xlab="Taille des séquences : n (log)",
     ylab="Temps (s, log)",
     main="Algo dp avec tas (C++) (log-log)")
abline(lm(log(time) ~ log(n), data=res4), col="red")
```










```r
par(mfrow=c(1,1))
```

```r
modele1 <- lm(log(time) ~ log(n), data = res1)
summary(modele1)
```

```
##
## Call:
## lm(formula = log(time) ~ log(n), data = res1)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.07501 -0.03999 -0.01205  0.03204  0.13738
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -7.40275    0.11781  -62.84 4.58e-12 ***
## log(n)       0.97660    0.02961   32.98 7.79e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.06544 on 8 degrees of freedom
## Multiple R-squared:  0.9927, Adjusted R-squared:  0.9918
## F-statistic:  1088 on 1 and 8 DF,  p-value: 7.792e-10
```

```r
modele2 <- lm(log(time) ~ log(n), data = res2)
summary(modele2)
```

```
##
## Call:
## lm(formula = log(time) ~ log(n), data = res2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.54452 -0.10201  0.03533  0.12664  0.35418
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -8.8011     0.5033  -17.49 1.17e-07 ***
## log(n)        2.4002     0.1265   18.98 6.16e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2795 on 8 degrees of freedom
## Multiple R-squared:  0.9783, Adjusted R-squared:  0.9755
## F-statistic: 360.1 on 1 and 8 DF,  p-value: 6.155e-08
```

```r
modele3 <- lm(log(time) ~ log(n), data = res3)
summary(modele3)
```

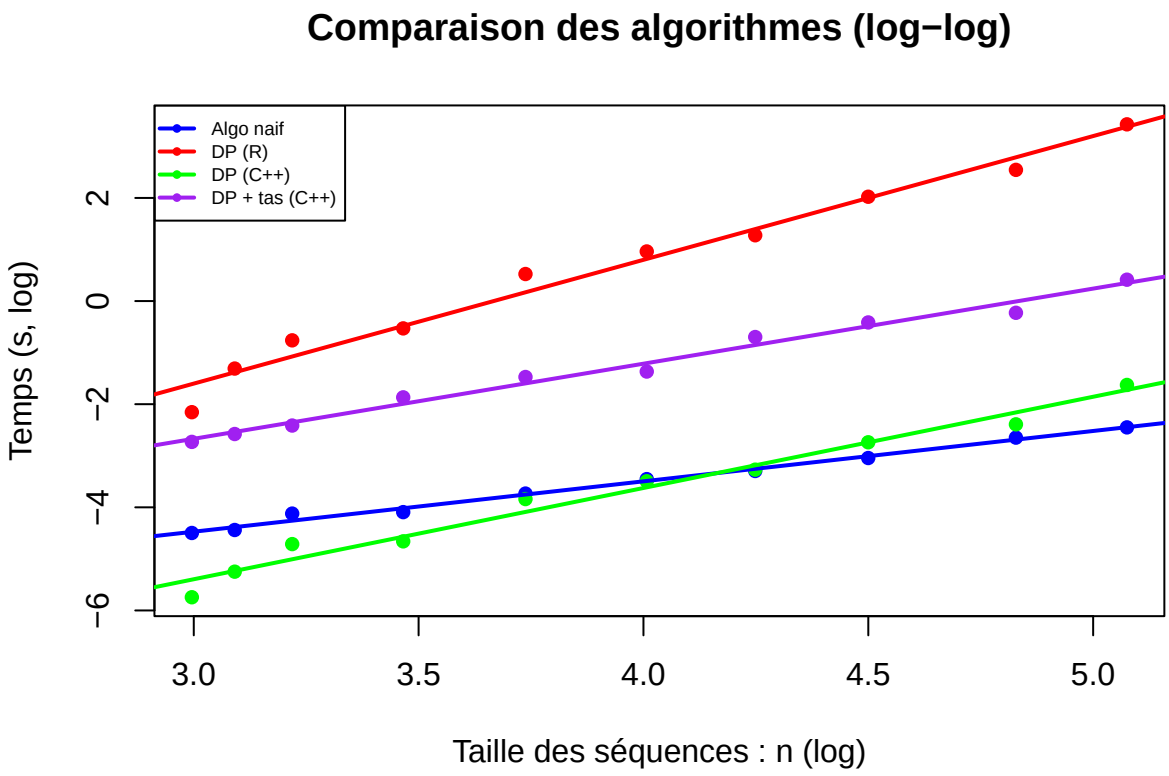
```
##
## Call:
## lm(formula = log(time) ~ log(n), data = res3)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.34126 -0.08803 -0.00596  0.11245  0.29447
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -10.70394    0.38049  -28.13 2.75e-09 ***
## log(n)        1.76961    0.09563   18.50 7.50e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2113 on 8 degrees of freedom
## Multiple R-squared:  0.9772, Adjusted R-squared:  0.9743
## F-statistic: 342.4 on 1 and 8 DF,  p-value: 7.497e-08
```

```r
modele4 <- lm(log(time) ~ log(n), data = res4)
summary(modele4)
```

```
##
## Call:
## lm(formula = log(time) ~ log(n), data = res4)
##
## Residuals:
```

```
##      Min       1Q   Median       3Q      Max
## -0.21927 -0.06027  0.01153  0.11081  0.15513
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -7.03946    0.24438  -28.81 2.28e-09 ***
## log(n)       1.45624    0.06142   23.71 1.07e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1357 on 8 degrees of freedom
## Multiple R-squared:  0.986,  Adjusted R-squared:  0.9842
## F-statistic: 562.1 on 1 and 8 DF,  p-value: 1.066e-08
```

```r
# Couleurs pour chaque algorithme
cols <- c("blue", "red", "green", "purple")

# Déterminer les limites de l'axe x et y pour inclure toutes les données
xlim <- range(log(res1$n), log(res2$n), log(res3$n), log(res4$n))
ylim <- range(log(res1$time), log(res2$time), log(res3$time), log(res4$time))

# Graphique vide pour préparer l'affichage
plot(NA, NA, xlim=xlim, ylim=ylim,
     xlab="Taille des séquences : n (log)",
     ylab="Temps (s, log)",
     main="Comparaison des algorithmes (log-log)")

# Ajouter les 4 séries de points
points(log(res1$n), log(res1$time), pch=16, col=cols[1])
points(log(res2$n), log(res2$time), pch=16, col=cols[2])
points(log(res3$n), log(res3$time), pch=16, col=cols[3])
points(log(res4$n), log(res4$time), pch=16, col=cols[4])

# Ajouter les droites de régression
abline(lm(log(time) ~ log(n), data=res1), col=cols[1], lwd=2)
abline(lm(log(time) ~ log(n), data=res2), col=cols[2], lwd=2)
abline(lm(log(time) ~ log(n), data=res3), col=cols[3], lwd=2)
abline(lm(log(time) ~ log(n), data=res4), col=cols[4], lwd=2)

# Ajouter la légende
legend("topleft",
       legend=c("Algo naif", "DP (R)", "DP (C++)", "DP + tas (C++)"),
       col=cols, pch=16, lwd=2,
       cex=0.6)
```



Les graphiques en échelle *log-log* montrent que chaque algorithme suit une relation de type $T(n) \propto n^{\alpha}$, ce que confirment les régressions linéaires dont les coefficients de détermination sont très élevés ($R^2 > 0.96$).
Les pentes estimées permettent de comparer directement la croissance du temps de calcul entre algorithmes.

- **Algorithme naïf (R)** : la pente (~0.95) indique une croissance presque linéaire. L'algorithme est très rapide mais ne garantit pas l'optimalité.
- **Programmation dynamique (R)** : la pente élevée (~2.3) montre une forte augmentation du temps de calcul. L'algorithme devient difficilement utilisable lorsque $n > 100$.
- **Programmation dynamique (C++)** : la pente plus faible (~1.7) reflète une complexité mieux contrôlée. L'implémentation est environ 100× plus rapide que celle en R.
- **Programmation dynamique avec tas (C++)** : la croissance du temps reste modérée avec une d'environ 1.35, même pour de grandes valeurs de n. Si on utiliserait des valeurs plus grandes que celle testées, le temps de cet algorithme serait moins important que celui d'une programmation dynamique standard.