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 5

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.

Asin	Score	Category	Sponsored
B00004ZCJI	4,4	Electronics	False
B00004ZCJJ	4,4	Electronics	False
B00079ULA8	4,1	Sports	False
B000CBSNRY	4,9	Toys_Games	False
B0001YR54E	4,5	Clothing	True

Id	Score	Groupe
1	4,4	Electronics, not_sponsored
2	4,4	Electronics, not_sponsored
3	4,1	Sports, not_sponsored
4	4,9	Toys_Games, not_sponsored
5	4,5	Clothing, sponsored

1.2.2. Formulation mathématique

La fonction objectif est :

$$\max_x \sum_{i=1}^{n} \sum_{p=1}^{5} w_p \, s_i \, x_{i,p}$$

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 :

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

- 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 :

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

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

$$\sum_{i \in \text{not sponsored}} \sum_{p=1}^{5} x_{i,p} \leq 4$$

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

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

```
## [1] 72.5
```

```
##   id score                  groupes position poids_position score_pondere
## 1 50   4.9     Beauty,Not_sponsored        1              5          24.5
## 2  3   4.8 Toys_Games,Not_sponsored        2              4          19.2
## 3 22   4.8         Beauty,Sponsored        3              3          14.4
## 4 25   4.8 Toys_Games,Not_sponsored        4              2           9.6
## 5 35   4.8          Books,Sponsored        5              1           4.8
```

```
## [1] 72.5
```

- 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.

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

La relation de récurrence est :

$$DP[p][\alpha] = max\{DP[p][\alpha], DP[p-1][\alpha'] + w_p s_i\}$$

Sous contraintes : * i n'a pas encore été utlisé * $\alpha = \alpha' + b(i)$ * $\alpha'_g \leq \alpha_g \quad \forall G$

avec

- p : la position
- $\alpha = (\alpha[1], \alpha[2], ..., c[G])$: vecteur des compteurs par groupe
- $DP[p][\alpha]$: meilleur score pour avoir sélectionné p éléments avec configuration α.
- $b[i] = (b[i]_1, b[i]_2, \ldots, b[i]_G)$: vecteur binaire de groupe où :

$$b[i]_g = \begin{cases} 1 & \text{si l'élément } i \text{ appartient au groupe } g \\ 0 & \text{sinon} \end{cases}$$

Exemple concret

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

Table 3: Tableau des éléments sélectionnés par l'algorithme ranking max

	Position	ID	Score	Groupe	Poids position	Score pondéré
50	1	50	4.9	Beauty , Not_sponsored	**5**	24.5
35	2	35	4.8	Books , Sponsored	**4**	19.2
3	3	3	4.8	Toys_Games , Not_sponsored	**3**	14.4
25	4	25	4.8	Toys_Games , Not_sponsored	**2**	9.6
22	5	22	4.8	Beauty , Sponsored	**1**	4.8

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 approximative 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,
   - les Id des éléments sélectionnés précédemment.
2. 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 tas et ne garde que "beam_size" meilleurs états (pruning).
3. 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 100** :

```
## $selected_items
##   index score                 groupes position poids_position score_pondere
## 1    50   4.9     Beauty,Not_sponsored        1              5          24.5
```

```
## 2    22   4.8         Beauty,Sponsored        2              4          19.2
## 3    35   4.8          Books,Sponsored        3              3          14.4
## 4    32   4.8     Toys_Games,Sponsored        4              2           9.6
## 5     3   4.8 Toys_Games,Not_sponsored        5              1           4.8
##
## $best_score
## [1] 72.5
##
## $total_items
## [1] 5
##
## $beam_size
## [1] 100
##
## $is_approximate
## [1] TRUE
##
## $final_counts
##   Electronics  Home_Kitchen        Sports         Books    Toys_Games
##             0             0             0             1             2
##      Clothing        Beauty  Pet_Supplies        Office    Automotive
##             0             2             0             0             0
## Not_sponsored
##             2
```

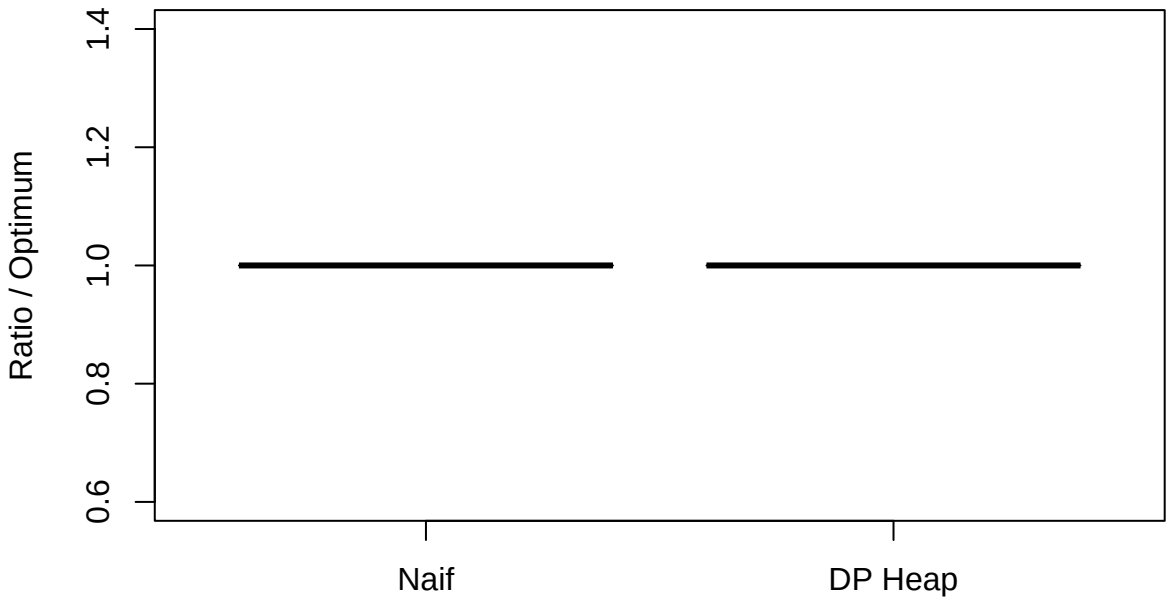
On reprend l'exemple précédent, avec un **beam de taille différente égale à 5000** :

```
## $selected_items
##   index score                 groupes position poids_position score_pondere
## 1    50   4.9     Beauty,Not_sponsored        1              5          24.5
## 2    32   4.8     Toys_Games,Sponsored        2              4          19.2
## 3    22   4.8         Beauty,Sponsored        3              3          14.4
## 4     3   4.8 Toys_Games,Not_sponsored        4              2           9.6
## 5    35   4.8          Books,Sponsored        5              1           4.8
##
## $best_score
## [1] 72.5
##
## $total_items
## [1] 5
##
## $beam_size
## [1] 5000
##
## $is_approximate
## [1] TRUE
##
## $final_counts
##   Electronics  Home_Kitchen        Sports         Books    Toys_Games
##             0             0             0             1             2
##      Clothing        Beauty  Pet_Supplies        Office    Automotive
##             0             2             0             0             0
## Not_sponsored
##             2
```

- 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 <- 100         # 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 n=200",
        ylab="Ratio / Optimum", col=c("red","blue"))
```



On constate que l'approche heuristique et la programmation dynamique avec tas fonctionnent. Ils trouvent les mêmes solutions que l'algorithme de programmation dynamique standard.

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(5, 4.8, 4.7, 4.6, 4, 3),
  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(5, 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   5.0     A,S        1              5          25.0
## 2  2   4.8     A,S        2              4          19.2
## 3  4   4.6     B,S        3              3          13.8
## 4  5   4.0     C,S        4              2           8.0
##
## $best_score
## [1] 66
```

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

```
## Resultat programmation dynamique
```

```r
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   5.0     A,S        1              5          25.0
## 2     3   4.7       A        2              4          18.8
## 3     4   4.6     B,S        3              3          13.8
## 4     5   4.0     C,S        4              2           8.0
## 5     6   3.0     C,S        5              1           3.0
##
## $best_score
## [1] 68.6
```

De même, si le paramètre beam_size est trop faible, la solution trouvée par l'algorithme de programmation dynamique avec tas n'est pas optimale :

```r
data <- data.frame(
  id = 1:6,
  score = c(5, 4.8, 4.7, 4.6, 4, 3),
  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(5, 1, by = -1)

cat("Resultat beam search\n")
```

```
## Resultat beam search
```

```r
resultat_beam_search <- ranking_max_dp_heap_cpp(data, 5, max_par_groupe, poids_positions, beam_size = 1)
```

```
## Warning in ranking_max_dp_heap_cpp(data, 5, max_par_groupe, poids_positions, :
## Aucun état valide à la position 5. Les contraintes sont peut-être trop
## restrictives.
```

```r
print(resultat_beam_search)
```

```
## $selected_items
##   index score groupes position poids_position score_pondere
## 1     1   5.0     A,S        1              5          25.0
## 2     2   4.8     A,S        2              4          19.2
## 3     4   4.6     B,S        3              3          13.8
## 4     5   4.0     C,S        4              2           8.0
##
## $best_score
## [1] 66
##
## $total_items
## [1] 4
##
## $beam_size
## [1] 1
##
```

```
## $is_approximate
## [1] TRUE
##
## $final_counts
## A B C S
## 2 1 1 4
```

```
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   5.0     A,S        1              5          25.0
## 2     3   4.7       A        2              4          18.8
## 3     4   4.6     B,S        3              3          13.8
## 4     5   4.0     C,S        4              2           8.0
## 5     6   3.0     C,S        5              1           3.0
##
## $best_score
## [1] 68.6
```

Remarque : Un beam_size égal à revient au même que l'algorithme naïf, on garde le meilleur état à chaque étape.

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
- $\bar{g}$: 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 : $O(G)$
  - Vérifier les contraintes : $O(G)$ comparaisons
  - Calculer le gain : $O(1)$
- Mise à jour des compteurs : $O(G)$

Coût pour la position p :

$$T_p = O((n - p + 1) \times G)$$

Coût total de la boucle :

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

Complexité temporelle

$$\boxed{T_{\text{naïf}}(n) = O(knG)}$$

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

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

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

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 : $O(nG)$
- Créer la matrice binaire `item_groups` : $O(nG^2)$

Étape 2 - Programmation dynamique

Structure : `DP[[p+1]][[key]]` avec `key = "c1,c2,...,cG"`

Nous obtenu ce calcul :

- $O(k)$: Boucle sur les positions `for (p in 1:k)`
- $O(k)$: Nombre d'états par position `for (key_prev in names(DP[[p]]))`
- $O(n)$: Boucle sur tous les candidats `for (i in 1:n)`
- $O(G)$: Calculer nouveaux compteurs `new_counts <- prev_counts + grp_i`
- $O(G)$: Vérifier contraintes `if (any(new_counts > max_cap))`

Étape 3 - Recherche du meilleur : $O(G)$

Complexité temporelle

$$\boxed{T_{\text{DP-R}}(n) = O(nk^2G^2)}$$

Meilleur cas : $\boxed{T(n) = O(nk^2)}$

Cas moyen : $\boxed{T(n) = O(nk^2\bar{g}^2)}$

Pire cas : $\boxed{T(n) = O(nk^2G^2)}$

3. Algorithme DP (C++)

Même logique que DP R, voici la complexité.

Complexité temporelle

$$\boxed{T_{\text{DP-R}}(n) = O(nk^2G^2)}$$

Meilleur cas : $\boxed{T(n) = O(nk^2)}$

Cas moyen : $\boxed{T(n) = O(nk^2\bar{g}^2)}$

Pire cas : $\boxed{T(n) = O(nk^2G^2)}$

4. Algorithme dynamique amélioré (C++)

Principe : Programmation dynamique avec hachage et beam pruning pour éviter l'explosion combinatoire.

Analyse de l'algorithme

Phase 1-4 - Prétraitement : $O(n \times \bar{g})$

- Parsing des groupes et construction de la matrice d'appartenance `belongs[n][G]`
- $\bar{g}$ = nombre moyen de groupes par élément

Phase 5 - Boucle DP principale

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

- Nombre d'états explorés : B (beam_size)
- Pour chaque état parent :
  - Reconstruction d'historique (Unicité) : $O(p \cdot G)$ (par le *backtracking* sur les p étapes précédentes)
- Pour chaque candidat n :
  - Vérifications par candidat :
  - Unicité : $O(1)$ (hash lookup sur l'historique reconstruit)
  - Contraintes : $O(G)$ (vérifier tous les groupes)
  - Copie d'état si valide : $O(G)$ (stockage de l'état allégé pour le *backtracking*)
- **Beam pruning** (si $|\text{DP curr}| > B$) :
  - Tri partiel (`nth_element`) : $O(|\text{DP curr}|)$
  - Conservation des B meilleurs états

Coût par position :

$$T_{\text{position}}(p) = O(BpG + BnG) = O(BG(n + p))$$

Coût total de la boucle :

$$T_{\text{boucle}} = \sum_{p=1}^{k} O(BG(n+p)) = O(BG(nk+k^2))$$

Complexité temporelle

$$\boxed{T_{\text{DP}}(n) = O(n\bar{g}) + O(BG(nk+k^2))}$$

Meilleur cas : $(G=1)$: $T(n) = O(B(nk+k^2))$

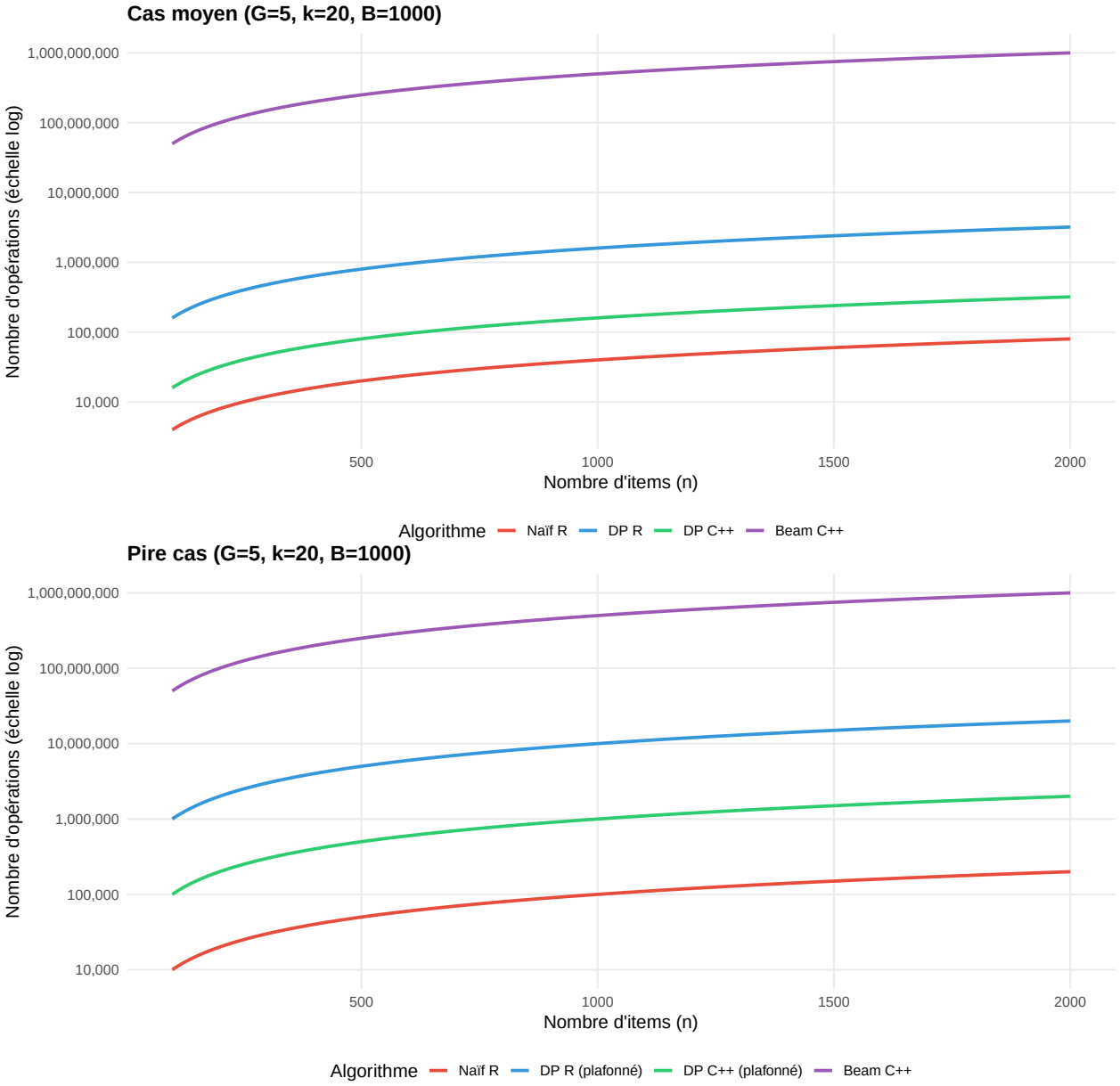
Cas moyen : $T(n) = O(B\bar{g}(nk+k^2))$

Pire cas : $T(n) = O(BG(nk+k^2))$

Tableau récapitulatif

Algorithme	Meilleur cas	Cas moyen	Pire cas
Naïf R	$O(kn)$	$O(kn\bar{g})$	$O(knG)$
DP R	$O(nk^2)$	$O(nk^2\bar{g^2})$	$O(nk^2G^2)$
DP C++	$O(nk^2)$	$O(nk^2\bar{g^2})$	$O(nk^2G^2)$
Beam C++	$O(B(nk+k^2))$	$O(B\bar{g}(nk+k^2))$	$O(BG(nk+k^2))$

Comparaison graphique



Observations

Cas moyen :

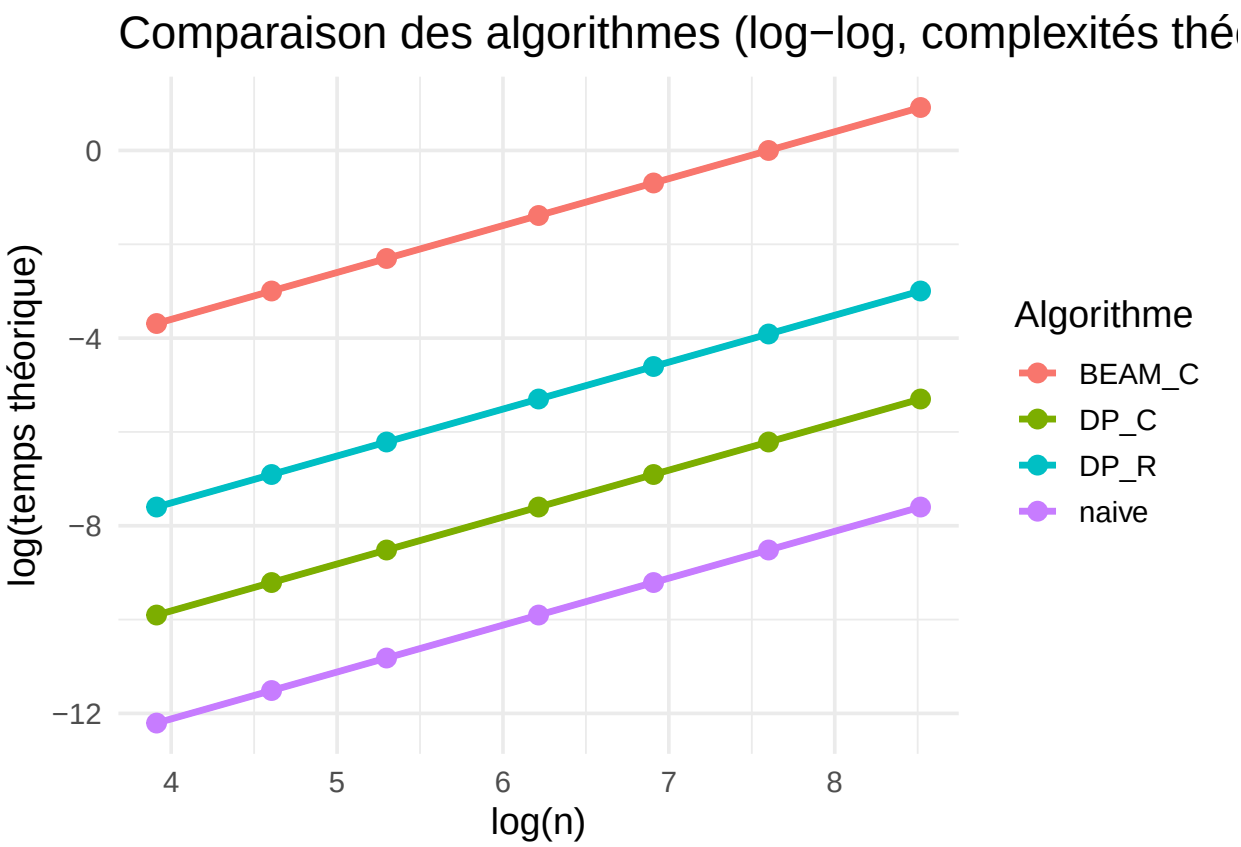
- Beam Search et DP C++ sont très compétitifs
- DP R significativement plus lent
- Naïf acceptable pour des petites instances

Pire cas :

- Explosion exponentielle des DP exact (R et C++)
- Beam Search reste linéaire et prévisible
- Seul algorithme viable pour grands problèmes avec $G \geq 6$

```r
# Paramètres de simulation
G <- 5
k <- 20
B <- 1000   # Beam size
n_values <- c(50, 100, 200, 500, 1000, 2000, 5000)

C1 <- 1e-9
C2 <- 1e-9
C3 <- 1e-9
C4 <- 1e-9

df <- data.frame(
  n = n_values,

  naive   = C1 * (k * n_values * G),
  DP_R    = C2 * (n_values * k * k * G * G),
  DP_C    = C3 * (n_values * k * k * G * G)/10,
  BEAM_C  = C4 * (B * n_values * k * (G + k))
)

library(dplyr)
library(tidyr)

df_long <- df %>%
  pivot_longer(-n, names_to = "algo", values_to = "time")

library(ggplot2)

ggplot(df_long, aes(x = log(n), y = log(time), color = algo)) +
  geom_point(size = 3) +
  geom_line(size = 1.2) +
  geom_smooth(method = "lm", se = FALSE, linewidth = 1) +
  labs(
    title = "Comparaison des algorithmes (log-log, complexités théoriques)",
    x = "log(n)",
    y = "log(temps théorique)",
    color = "Algorithme"
  ) +
  theme_minimal(base_size = 14)
```

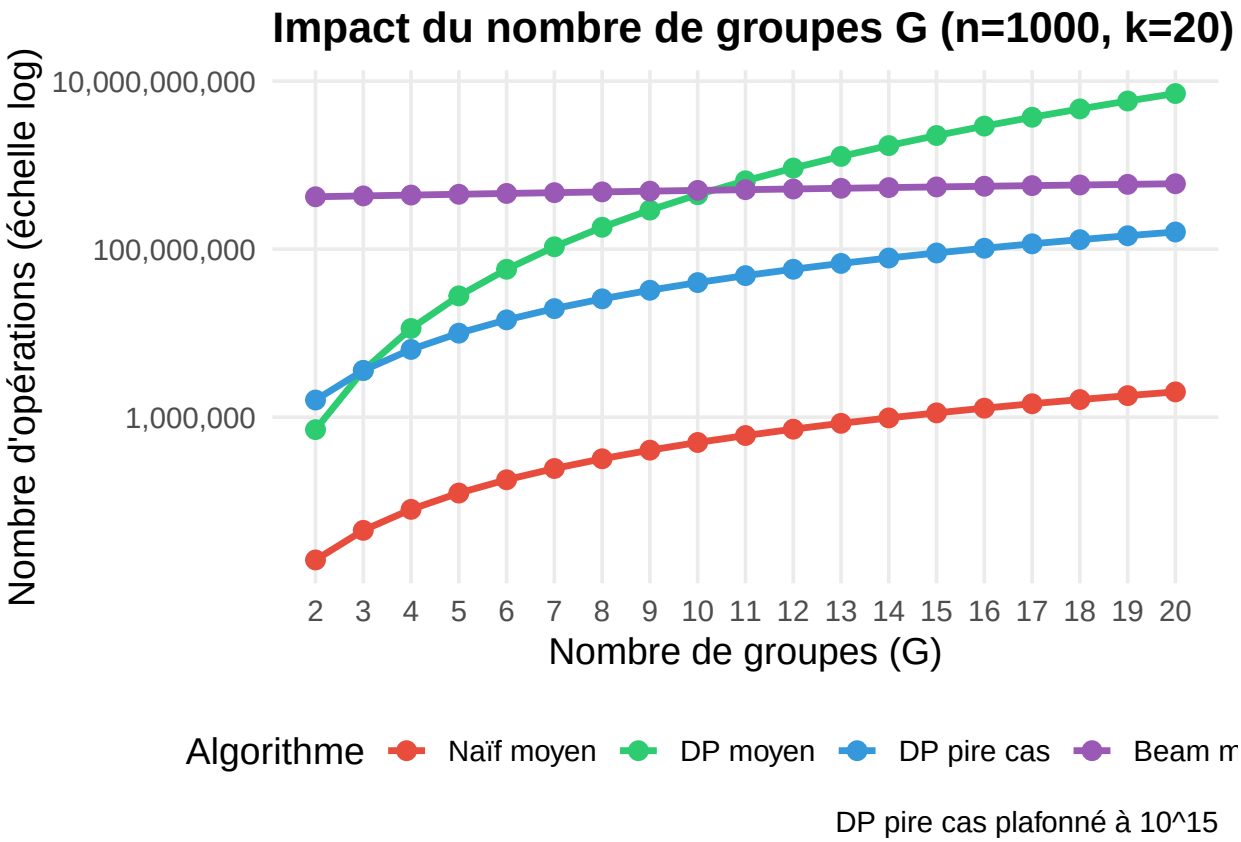
```
## `geom_smooth()` using formula = 'y ~ x'
```



Impact du nombre de groupes G

```r
library(ggplot2)
library(dplyr)
library(tidyr)
library(scales)

# Paramètres fixes
n <- 1000
k <- 20
B <- 1000

G_values <- 2:20

# Moyennes approximatives
g_bar <- G_values / 2                # moyenne
g2_bar <- G_values^2 / 3             # moyenne des carrés

df_impact <- data.frame(
  G = G_values,

  # Naïf
  Naive_moyen = k * n * g_bar^2,
  Naive_pire  = k * n * G_values^2,

  # DP
```

```r
  DP_moyen = n * k^2 * g2_bar^2,
  DP_pire  = n * k^2 * G_values^2,

  # Beam
  Beam_moyen = B * n * k * (g_bar + k),
  Beam_pire  = B * n * k * (G_values + k)
)

# Pire cas plafonné pour lisibilité
df_impact$DP_pire_display <- pmin(df_impact$DP_pire, 1e15)

df_long <- df_impact %>%
  select(G, Naive_moyen, DP_moyen, DP_pire_display, Beam_moyen) %>%
  pivot_longer(-G, names_to = "algo", values_to = "operations") %>%
  mutate(
    algo = factor(algo,
      levels = c("Naive_moyen", "DP_moyen", "DP_pire_display", "Beam_moyen"),
      labels = c("Naïf moyen", "DP moyen", "DP pire cas", "Beam moyen")
    )
  )

ggplot(df_long, aes(x = G, y = operations, color = algo)) +
  geom_line(size = 1.2) +
  geom_point(size = 3) +
  scale_y_log10(labels = comma) +
  scale_x_continuous(breaks = G_values) +
  scale_color_manual(values = c(
    "Naïf moyen" = "#E74C3C",
    "DP moyen" = "#2ECC71",
    "DP pire cas" = "#3498DB",
    "Beam moyen" = "#9B59B6"
  )) +
  theme_minimal(base_size = 14) +
  theme(
    legend.position = "bottom",
    plot.title = element_text(face = "bold", size = 16),
    panel.grid.minor = element_blank()
  ) +
  labs(
    title = sprintf("Impact du nombre de groupes G (n=%d, k=%d)", n, k),
    x = "Nombre de groupes (G)",
    y = "Nombre d'opérations (échelle log)",
    color = "Algorithme",
    caption = "DP pire cas plafonné à 10^15"
  )
```



On observe que pour un petit nombre de groupe l'algorithme heuristique est le meilleur et le beam est le pire. Pour un grand nombre de groupes, le beam devient meilleur que le dynamique simple car son nombre d'opérations reste constant.

7. Temps de calcul

```r
set.seed(123)

n <- c(20,22,25,32,42,55,70,90,125,160,200,250)
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] 0.003609896 0.003437996 0.019201994 0.004777908 0.006111145 0.007977962
##  [7] 0.010352135 0.017920017 0.017534018 0.027140856 0.028094053 0.046063900
```

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

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

```
temps_moyens2
```

```
##  [1]  0.3646829  0.2419310  0.3270040  1.2680271  0.9719419  2.3217781
##  [7]  5.7174339  6.7292478 13.3314462 29.5019341 31.4706728 43.4656711
```

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

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

```
temps_moyens3
```

```
##  [1] 0.00819993 0.01912498 0.01051283 0.02530003 0.01736307 0.04020000
##  [7] 0.05965304 0.07304192 0.15401316 0.24496102 0.29209805 0.36415410
```

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

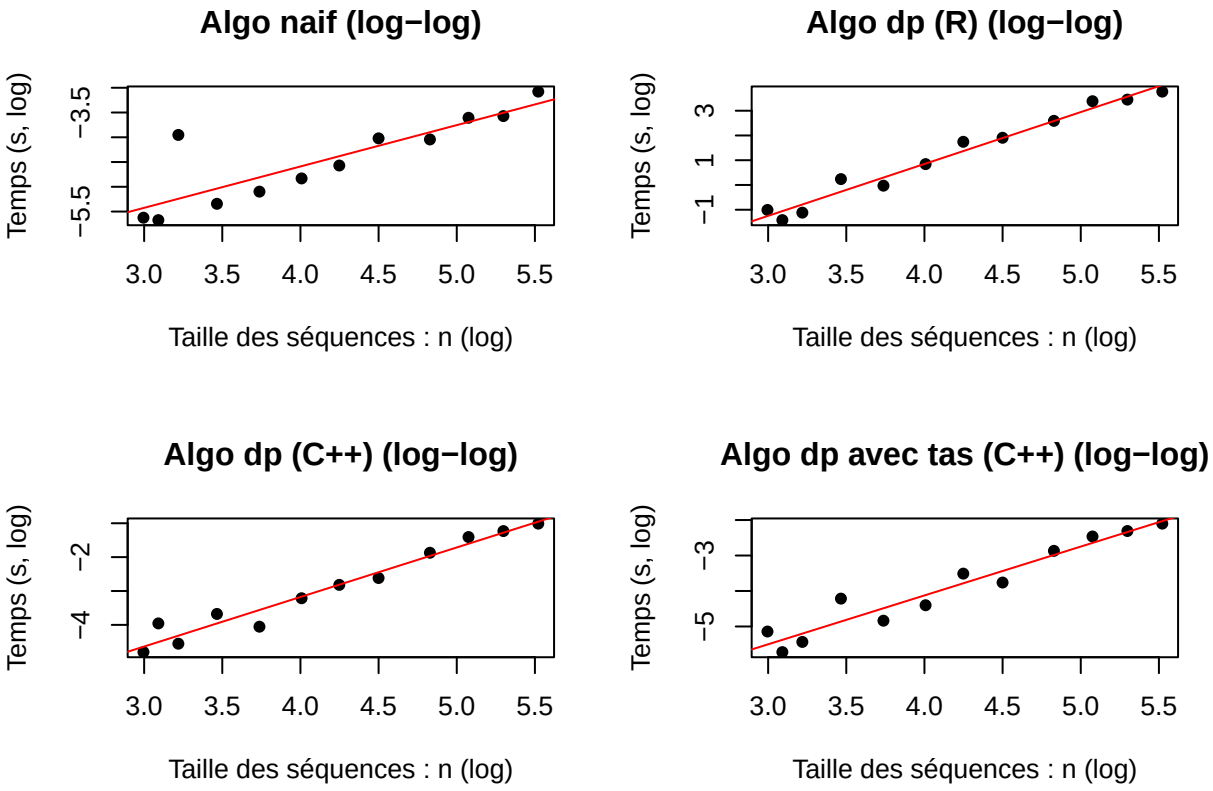
```
## Temps pour l'algorithme de programmation dynamique avec tas (en C++)
```

```
temps_moyens4
```

```
##  [1] 0.005845785 0.003273010 0.004360199 0.014780998 0.007930040 0.012263060
##  [7] 0.029885054 0.023252964 0.056787014 0.085162878 0.099344969 0.122392178
```

```
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")
```




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

```
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
## -0.3239 -0.2579 -0.1672  0.1011  1.2894
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -7.9310     0.6781 -11.696 3.72e-07 ***
## log(n)        0.8354     0.1595   5.237 0.000381 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4684 on 10 degrees of freedom
## Multiple R-squared:  0.7328, Adjusted R-squared:  0.7061
## F-statistic: 27.42 on 1 and 10 DF,  p-value: 0.0003805
```

```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
## -0.36305 -0.28078 -0.00815  0.25593  0.50830
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -7.5337     0.4514  -16.69 1.25e-08 ***
## log(n)        2.0956     0.1062   19.74 2.44e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3118 on 10 degrees of freedom
## Multiple R-squared:  0.975,  Adjusted R-squared:  0.9725
## F-statistic: 389.5 on 1 and 10 DF,  p-value: 2.443e-09
```

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

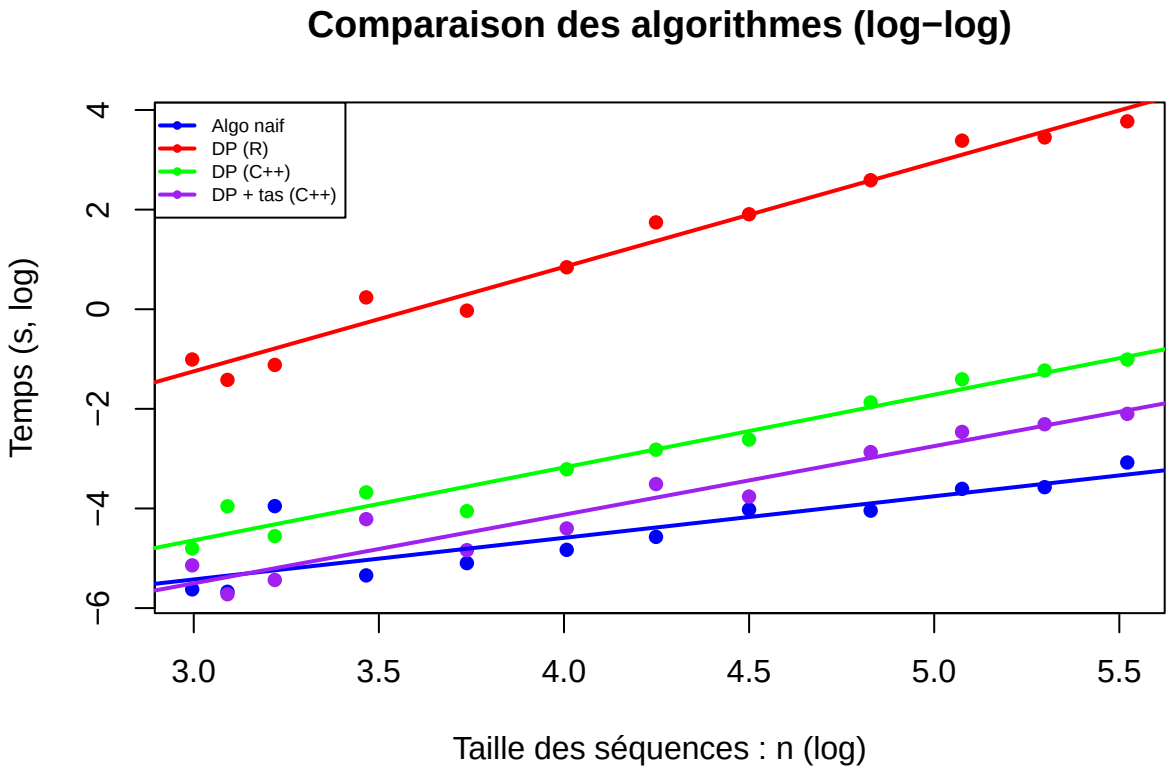
```
##
## Call:
## lm(formula = log(time) ~ log(n), data = res3)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.49303 -0.16141 -0.02685  0.12050  0.54924
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -9.0263     0.4068  -22.19 7.76e-10 ***
## log(n)        1.4624     0.0957   15.28 2.92e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.281 on 10 degrees of freedom
## Multiple R-squared:  0.9589, Adjusted R-squared:  0.9548
## F-statistic: 233.5 on 1 and 10 DF,  p-value: 2.925e-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
## -0.35064 -0.29563 -0.02111  0.20425  0.64651
##
```

```
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -9.6334     0.4936  -19.52 2.72e-09 ***
## log(n)        1.3770     0.1161   11.86 3.26e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.341 on 10 degrees of freedom
## Multiple R-squared:  0.9336, Adjusted R-squared:  0.927
## F-statistic: 140.7 on 1 and 10 DF,  p-value: 3.261e-07
```

```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 (~0.93) indique une croissance presque linéaire. L'algorithme est très rapide mais ne garantit pas l'optimalité.
- **Programmation dynamique (R)** : la pente élevée (~1.96) 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.
- **Programmation dynamique avec tas (C++)** : la croissance du temps reste modérée avec une pente d'environ 1.27, même pour de grandes valeurs de n. Cependant, comme l'algorithme naïf, l'optimalité n'est pas garantit.

Conclusion

Choix de l'algorithme selon le contexte

Contexte	Algorithme recommandé	Justification
Petits problèmes (n<100, G<3, k<10)	**Naïf R** ou **DP C++** ou **Beam C++**	Rapide, optimal (garanti pour DP C++)
Moyens problèmes (n<1000, G<5, k<20)	**Naïf R** ou **DP C++** ou **Beam C++**	Optimal (garanti pour DP C++), très bonne performance
Grands problèmes (n>1000 ou G>5)	**Naïf R**	Le plus rapide, quasi-optimal

Contexte	Algorithme recommandé	Justification
Recherche d'optimum prouvé	**DP C++** si faisable	Exact et plus rapide que **DP R**