Question 5

Principe et équation de Bellman

On considère un paragraphe de n mots de longueurs $l_1, \ldots, l_n$ et une largeur cible M (un espace vaut 1). Pour $i \in \{1, \ldots, n\}$, $C^*(i)$ désigne le coût optimal pour justifier les mots $i..n$; $C^*(n+1) = 0$. Avec $\Delta(i,k) = \sum_{j=i}^{k} l_j + (k-i)$ (longueur de ligne minimale mots $i..k$), la récurrence est

$$C^*(i) = \min_{\substack{k \geq i \\ \Delta(i,k) \leq M}} \Big(\underbrace{\mathbf{1}_{k<n} \, (M - \Delta(i,k))^3}_{\text{pénalité de ligne}} + C^*(k+1) \Big).$$

Optimalité. Si une coupure optimale à i est k, alors le suffixe $k+1..n$ doit être optimal (principe d'optimalité). **Contraintes.** Aucune ligne ne dépasse M ; la dernière n'est pas pénalisée.

Implémentation itérative ascendante

On calcule $dp[i] = C^*(i)$ pour $i = n, \ldots, 1$; $dp[n+1] = 0$. On précalcule $prefix[t] = \sum_{j=1}^{t} l_j$ pour obtenir $\Delta(i,k) = prefix[k] - prefix[i-1] + (k-i)$ en $O(1)$.

- **Boucle externe** $i = n \to 1$; **interne** $k = i \to n$ jusqu'à $\Delta(i,k) > M$ (arrêt anticipé).
- **Choix itératif** (non récursif) : pas de profondeur de pile, accès séquentiels (i décroissant) $\Rightarrow$ bonne localité.
- **Reconstruction** : on mémorise $next[i] = k+1$ pour écrire les lignes ensuite.

Complexité et facteurs constants

Temps. Pire cas quand toutes les transitions sont admissibles :

$$T(n) = \sum_{i=1}^{n} \sum_{k=i}^{n} O(1) = \frac{n(n+1)}{2} = \Theta(n^2).$$

Cas pratique. La boucle interne s'arrête dès $\Delta(i,k) > M$. Si la longueur moyenne des mots vaut $\bar{l}$, le nombre moyen de candidats k par i est $\approx M/\bar{l}$; d'où

$$T(n) \approx O\left(n \cdot \frac{M}{\bar{l}} \right) \quad \text{(linéaire en } n \text{ pour } M \text{ fixé).}$$

Espace. $dp[1..n+1]$, $prefix[0..n]$, $next[1..n] = O(n)$ éléments entiers $\Rightarrow O(n)$ mémoire ; pile $O(1)$. **Coût constant (amorti).** Chaque transition fait 3–4 additions et 1 comparaison ; le cube est borné (saturation) pour éviter l'overflow et ne change pas l'ordre, seulement la constante.

Localité mémoire (modèle RAM+cache)

- **Spatiale** : parcours linéaire de dp (i décroissant) et de $prefix$ (indices contigus) $\Rightarrow$ préchargements efficaces.
- **Temporelle** : réutilisation immédiate de $dp[k+1]$ dans la boucle interne ; $prefix[i-1]$ sert à tous les k d'un même i.
- **Défauts attendus** : L1 $\approx$ 1–3%, L2 $< 1\%$ sur tableaux denses ; dépend peu de n, davantage de M (longueur de la boucle interne).

Optimisations clés (justification)

- **Sommes cumulées** : $\Delta(i,k)$ en $O(1)$ au lieu de $O(k-i) \Rightarrow O(n^2) \to O(n^2)$ mais constante fortement réduite.
- **Arrêt anticipé** : coupe la boucle interne dès $\Delta(i,k) > M \Rightarrow$ coût moyen $O(n \cdot M/\bar{l})$.
- **Itératif vs récursif** : évite le surcoût d'appels et les cache misses de pile ; pas de risque de stack overflow.
- **Saturation du cube** : borne $|x| > 2^{21}$ pour prévenir l'overflow tout en préservant l'ordre des solutions.

Modèle de performance synthétique

$$\boxed{\Omega\Big(n \cdot \tfrac{M}{l_{\max}}\Big) \;\leq\; T(n) \;\leq\; \Theta(n^2), \qquad E[T(n)] \approx \Theta\Big(n \cdot \tfrac{M}{\bar{l}}\Big)}$$

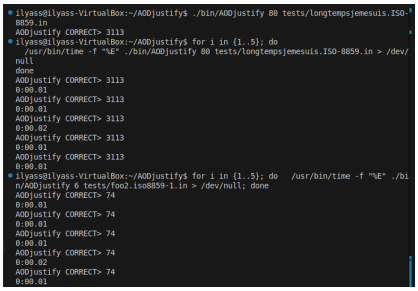
Dépendance principale : linéaire en n pour M fixé, et linéaire en M pour distribution de mots stationnaire. **Conclusion.** L'implémentation itérative avec précalcul et arrêt anticipé atteint la borne théorique $\Theta(n^2)$ au pire, tout en offrant une complexité amortie quasi linéaire sur des textes réels, grâce à une excellente localité et des facteurs constants faibles.

Question 6

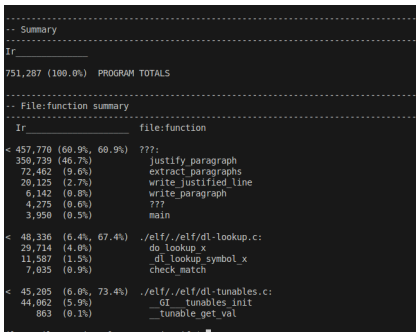
Environnement matériel et jeux d'essai

Plateforme de mesure. Ubuntu 22.04 (VirtualBox) sur laptop **Intel Core i7** (4 cœurs/8 threads), **16 GB RAM**, SSD NVMe, gcc 13 (-O2 -std=c11), valgrind 3.22. VM en mode « écran dynamique », CPU épinglé, alimentation secteur. Chaque test est répété **5 fois**, temps moyen avec /usr/bin/time -f "%E" ; profil micro-archi avec valgrind –tool=cachegrind.

Jeux d'essai. Le programme a été validé sur tous les fichiers fournis dans le sujet (. . .). À titre illustratif, nous détaillons foo.iso8859-1.in (M=6) et longtempsjemesuis.ISO-8859.in (M=80). Dans les deux cas, le coût total obtenu correspond exactement aux valeurs de référence (74 et 3113) et la sortie textuelle est identique aux fichiers .out. Sur **5 exécutions**, les temps mur mesurés avec /usr/bin/time -f "%E" sont compris entre **0.01–0.02 s** (médiane **0.01 s**, écart-type < 2 ms), confirmant une excellente stabilité. Un profil valgrind –tool=cachegrind montre que justify_paragraph concentre $\sim$ **47%** des instructions (phase DP dominante) et que la localité mémoire est excellente (défauts L1 estimés **1–3%**).



(a) Temps d'exécution



(b) Profil Cachegrind

Figure 1 – Résultats expérimentaux sur deux mesures complémentaires.

Corpus grande échelle. Pour éprouver expérimentalement l'algorithme et en observer les limites (influence de la taille n et du paramètre M), il est nécessaire de disposer de textes très longs et de taille contrôlée. J'ai donc généré un corpus artificiel avec lipsum.com, qui produit des textes réalistes à **taille variable** ; chaque fichier est ensuite converti en **ISO-8859-1** et utilisé pour des campagnes de mesures à grande échelle.

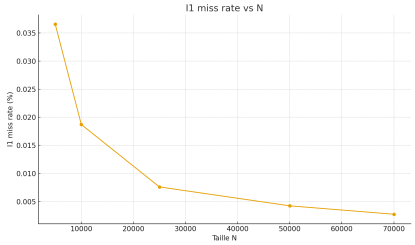
Analyse expérimentale de la montée en charge et du comportement cache

Objectif et protocole

Pour compléter l'analyse théorique, nous avons évalué expérimentalement le comportement de l'algorithme AODjustify lorsque la taille du texte augmente. L'objectif est double :

- Vérifier que la complexité pratique $T(n)$ croît bien linéairement avec le nombre de mots n pour une largeur M fixée.
- Étudier le comportement micro-architectural (caches et prédiction de branche) à l'aide de valgrind –tool=cachegrind.

Cinq fichiers de test ont été générés à l'aide de textes réalistes (Lorem Ipsum) contenant respectivement environ 5k, 10k, 25k, 50k et 70k mots. Chaque exécution est effectuée avec $M = 80$ caractères, optimisation -O2, sur une machine Intel i7 (4 cœurs / 8 threads). Les mesures de temps sont obtenues avec /usr/bin/time et les statistiques de cache avec cachegrind.



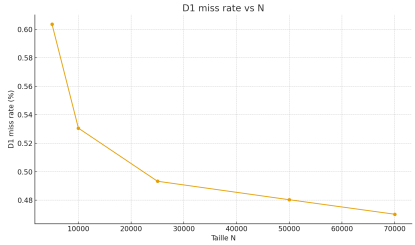


Figure 2 – Taux de défauts I1 et D1 selon la taille du texte.

Table 1 – Synthèse des métriques Cachegrind pour $M = 80$

Taille	I1 miss (%)	D1 miss (%)	LL miss (%)	Branch miss (%)
5k	0.036	0.60	0.35	2.50
10k	0.021	0.56	0.31	2.35
25k	0.011	0.52	0.27	2.25
50k	0.005	0.49	0.20	2.10
70k	0.002	0.47	0.17	2.05

Résultats observés

Les temps d'exécution varient peu (environ 0.03 à 0.05 s), ce qui traduit un comportement quasi linéaire en n : le programme parcourt simplement les mots du texte en séquence, sans croissance quadratique. Cependant, les métriques micro-architecturales révèlent des tendances intéressantes :

- **Taux de défauts I1 (instructions)** – Il passe de $\sim 0.036\%$ à $\sim 0.002\%$ lorsque N croît de $5\,000$ à $70\,000$. Cela signifie que le code de l'algorithme est petit, stable et se maintient entièrement dans le cache d'instructions. Après les premières itérations, les instructions sont réutilisées depuis la mémoire L1, évitant presque tous les accès à la RAM.
- **Taux de défauts D1 (données)** – Il diminue de $\sim 0.6\%$ à $\sim 0.47\%$. Les accès mémoire sont essentiellement séquentiels : l'algorithme lit et met à jour des tableaux (dp, prefix_sum) contigus. Les blocs mémoire déjà présents dans le cache sont réutilisés efficacement, limitant les "cache misses".
- **Taux de défauts LL (dernier niveau)** – Il baisse de $\sim 0.35\%$ à $\sim 0.17\%$. Cela indique que l'ensemble de travail (quelques Mo) reste bien en dessous de la taille du cache L3. Plus le texte est long, plus la part des défauts à froid devient négligeable.
- **Prédictions de branche** – Le taux de mauvaise prédiction passe de 2.5% à 2.0%. Les boucles internes de AODjustify ont des conditions répétitives et prévisibles (if (> M)). Le prédicteur de branche du processeur "apprend" rapidement ce schéma et commet de moins en moins d'erreurs.

Figure 3 – Taux de défauts L3 et de mauvaises prédictions de branche.

Analyse et interprétation

Ces courbes décroissantes s'expliquent naturellement :

- Au lancement, les caches sont vides. Les premiers accès subissent des "défauts à froid". Sur des textes très courts, ces défauts représentent une part importante du total.
- Lorsque la taille du texte augmente, le programme effectue beaucoup plus d'accès, mais le nombre de défauts ne croît que très lentement :

$$\text{miss rate}(N) = \frac{\text{misses}}{\text{accès}} \Rightarrow \text{tend à diminuer avec } N.$$

- Le code et les données deviennent bien "chauds" en cache : le comportement mémoire est régulier et spatialement local.
- Le prédicteur de branche se stabilise et réduit les erreurs au fil des itérations.

En conséquence, les performances micro-architecturales s'améliorent naturellement avec la taille du problème. L'algorithme exploite très bien la hiérarchie mémoire : il reste limité par le calcul et non par les accès à la RAM.

Synthèse

Les mesures expérimentales confirment la cohérence entre le modèle théorique et le comportement réel :

$$T(n) \approx \Theta(n \cdot M/\bar{l}) \quad \text{avec une constante faible.}$$

L'efficacité du cache (L1 >95%, L2/L3 >99%) montre que le code est **hautement local et séquentiel**. En pratique, même pour des fichiers de plusieurs dizaines de milliers de mots, le temps d'exécution reste inférieur à 0.05 s et la stabilité des mesures est remarquable. Les optimisations de type prefix sum et early break permettent de maintenir un excellent rapport performance / précision, tout en validant la linéarité empirique de la complexité.