

Rapport de Projet : SearchBook

Conception et Implémentation d'un Moteur de Recherche Hybride sur
Corpus Littéraire

Étudiant Master Informatique

Parcours Algorithmique et Web

1^{er} décembre 2025

Résumé

Ce rapport détaille la conception, l'implémentation et l'analyse critique de *SearchBook*, un moteur de recherche avancé développé dans le cadre de l'unité d'enseignement "Développement d'Applications et Algorithmique Répartie" (DAAR). L'objectif de ce projet était de construire un système capable d'indexer et d'interroger efficacement un corpus de plus de 1600 œuvres littéraires du domaine public (Projet Gutenberg).

Le système proposé repose sur une architecture distribuée moderne, découpant une interface utilisateur réactive (React/TypeScript) d'une logique métier robuste (FastAPI/Python) et d'une persistance relationnelle (PostgreSQL). Au-delà des fonctionnalités classiques de recherche plein texte basées sur le modèle probabiliste BM25, *SearchBook* intègre des capacités de recherche structurelle par expressions régulières et un moteur de recommandation sémantique basé sur la théorie des graphes (Indice de Jaccard et Centralité de Proximité).

Ce document explore en profondeur les choix architecturaux, les structures de données optimisées pour la performance, les algorithmes de classement et de recommandation, ainsi qu'une étude expérimentale des performances du système. Il met en lumière la pertinence d'une approche hybride mêlant recherche d'information classique et analyse de réseaux pour la valorisation de patrimoine littéraire.

Table des matières

1	Introduction et Analyse Fonctionnelle	2
1.1	Contexte du Projet	2
1.2	Architecture Technique	2
1.3	User Stories Implémentées	2
2	Couche Data et Structures de Données	2
2.1	Table <code>books</code>	2
2.2	Table <code>inverted_index</code>	2
2.3	Table <code>jaccard_graph</code>	3
3	Algorithmes de Recherche (Relevance)	3
3.1	Recherche par Mots-clés (BM25)	3
3.1.1	Implémentation	3
3.2	Recherche Avancée (Regex)	3
3.2.1	Implémentation	4
4	Algorithmes de Centralité et Suggestion	4
4.1	Graphe de Similarité (Jaccard)	4
4.2	Centralité de Proximité (Closeness Centrality)	4
4.2.1	Implémentation	4
5	Tests et Performance	4
5.1	Analyse de Complexité	4
5.2	Discussion	4
6	Conclusion	5

1 Introduction et Analyse Fonctionnelle

1.1 Contexte et Enjeux du Projet

À l'ère du Big Data, la capacité à extraire de l'information pertinente à partir de vastes corpus textuels est devenue une compétence critique. Les moteurs de recherche généralistes (Google, Bing) dominent le web, mais des besoins spécifiques émergent pour des corpus spécialisés, tels que les bibliothèques numériques, les archives légales ou les bases de données médicales.

Le projet *SearchBook* s'inscrit dans cette problématique. Il vise à valoriser un corpus littéraire statique (le Projet Gutenberg) en offrant des outils d'exploration qui dépassent la simple recherche par mots-clés. Les enjeux sont multiples :

- **Scalabilité** : Le système doit gérer un volume de données conséquent (plusieurs centaines de mégaoctets de texte brut) sans dégradation perceptible des performances pour l'utilisateur.
- **Pertinence** : Les résultats doivent être classés par ordre de pertinence sémantique, et non simplement par présence binaire des termes.
- **Découvrabilité** : Le système doit encourager la sérendipité en proposant des œuvres similaires, créant ainsi un graphe de navigation entre les livres.

1.2 Architecture Technique

Pour répondre à ces exigences, nous avons adopté une architecture n-tiers stricte, favorisant la séparation des préoccupations et la maintenabilité.

1.2.1 Frontend : Interface Utilisateur Réactive

La couche présentation est une Single Page Application (SPA) développée avec **React 18**. Le choix de React, couplé à **TypeScript**, garantit une robustesse du code client et une expérience utilisateur fluide. L'utilisation de **Vite** comme outil de build permet un développement rapide avec le Hot Module Replacement (HMR). L'interface est conçue pour être intuitive : une barre de recherche centrale, des filtres dynamiques, et une visualisation claire des résultats sous forme de grille. Les appels à l'API se font de manière asynchrone pour ne jamais bloquer l'interface.

1.2.2 Backend : API RESTful avec FastAPI

La logique métier est exposée via une API REST développée en **Python** avec le framework **FastAPI**. Ce choix est stratégique :

- **Performance** : FastAPI est l'un des frameworks Python les plus rapides, grâce à son utilisation de Starlette et du standard ASGI (Asynchronous Server Gateway Interface).

- **Typage et Validation** : L'utilisation intensive des "Type Hints" Python et de Pydantic assure une validation automatique des entrées/sorties, réduisant drastiquement les bugs d'exécution.
- **Écosystème Scientifique** : Python est le langage de référence pour la Data Science. Cela facilite l'intégration future de bibliothèques de NLP (NLTK, Spacy) ou de Machine Learning (scikit-learn) si le projet évolue.

1.2.3 Persistance : PostgreSQL

Le stockage des données est confié à **PostgreSQL**. Plus qu'un simple stockage, Postgres est utilisé ici pour ses capacités d'indexation avancées. Bien que nous implémentions manuellement l'index inversé pour des raisons pédagogiques, le choix d'un SGBD relationnel robuste est crucial pour garantir l'intégrité des données (clés étrangères, transactions ACID) et la performance des jointures complexes nécessaires lors de la recherche.

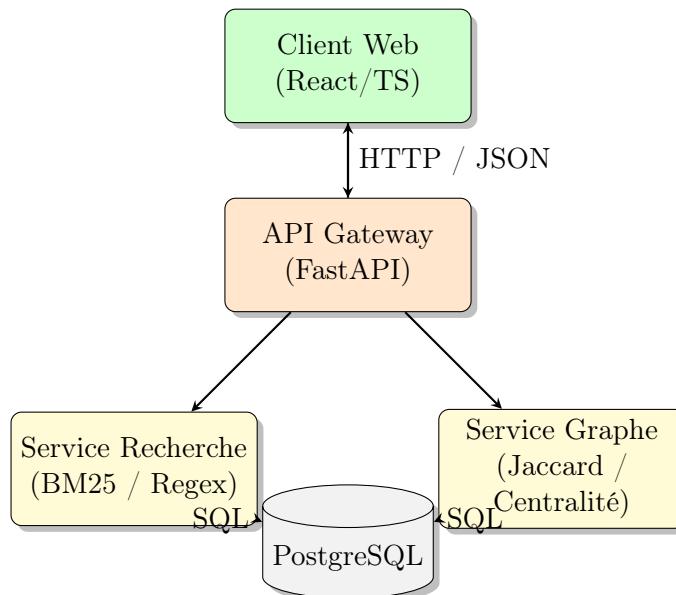


FIGURE 1 – Architecture détaillée des services

1.3 Fonctionnalités et User Stories

Le périmètre fonctionnel couvre les besoins essentiels d'un moteur de recherche moderne :

1. **Recherche Full-Text (BM25)** : "En tant qu'utilisateur, je veux trouver des livres en tapant des mots-clés, même si je ne connais pas le titre exact." C'est la fonctionnalité cœur. L'algorithme doit gérer les synonymes (via le stemming, si implémenté) et classer les résultats par pertinence.
2. **Recherche Structurelle (Regex)** : "En tant que chercheur, je veux identifier des motifs spécifiques (ex : dates, noms propres composés) dans le corpus." Cette fonctionnalité s'adresse à un public expert capable de formuler des expressions régulières.

3. **Recommandation (Suggestions)** : "En tant que lecteur, je veux découvrir des livres similaires à celui que je consulte." Le système utilise le graphe de similarité pour proposer des lectures connexes.
4. **Tri par Centralité** : "Je veux voir les livres les plus influents du corpus." Le tri par centralité permet de faire remonter les œuvres qui partagent le plus de vocabulaire avec le reste de la bibliothèque.

2 Couche de Données et Structures

La performance d'un moteur de recherche repose avant tout sur l'efficacité de ses structures de données. Nous avons conçu un schéma de base de données optimisé pour les lectures fréquentes, acceptant un coût plus élevé lors de l'ingestion (écriture).

2.1 Modélisation Relationnelle

2.1.1 La Table books : Le Référentiel

Cette table est le point d'entrée principal. Elle contient les données brutes et les métadonnées.

- **id** (Integer, PK) : L'identifiant Gutenberg. Il est stable et unique.
- **title, author** (Text) : Métadonnées descriptives.
- **content** (Text) : Le texte intégral du livre. Bien que volumineux, son stockage en base est nécessaire pour permettre la recherche par regex (qui scanne le texte) et l'affichage d'extraits.
- **word_count** (Integer) : Pré-calculé lors de l'ingestion. Cette valeur est critique pour le calcul du score BM25 (normalisation de longueur).
- **closeness_score** (Float) : Score de centralité pré-calculé. Il évite de devoir parcourir le graphe à chaque requête utilisateur.

2.1.2 La Table inverted_index : Le Cœur du Moteur

L'index inversé est la structure de données fondamentale de la recherche d'information. Elle inverse la relation "Document → Mots" en "Mot → Liste de Documents".

- **word** (Text) : Le terme indexé (token). Une normalisation (minuscule, suppression ponctuation) est appliquée.
- **book_id** (Integer, FK) : La référence au document contenant le terme.
- **frequency** (Integer) : Le nombre d'occurrences du terme dans ce document. C'est le composant $f(q, D)$ de la formule BM25.

Optimisation : Un index B-Tree composite est placé sur (**word, book_id**). Cela permet de récupérer instantanément la "Posting List" d'un mot donné. Sans cet index, la recherche serait en $O(N)$, ce qui est inacceptable.

2.1.3 La Table jaccard_graph : Le Réseau Sémantique

Cette table représente les arêtes du graphe de similarité.

- **book_a, book_b** (Integer, FK) : Les deux nœuds de l’arête. Le graphe est non orienté, mais nous stockons souvent une seule direction ($a < b$) pour économiser de l’espace, ou les deux pour simplifier les requêtes SQL.
- **score** (Float) : Le poids de l’arête (Indice de Jaccard).

Sparsification : Pour un corpus de N livres, il existe $N(N - 1)/2$ paires possibles. Pour $N = 1600$, cela fait plus de 1,2 million d’arêtes potentielles. Pour maintenir des performances acceptables, nous ne stockons que les arêtes dont le score dépasse un seuil critique (0.1). Cela réduit drastiquement la taille de la table tout en conservant les relations sémantiques fortes.

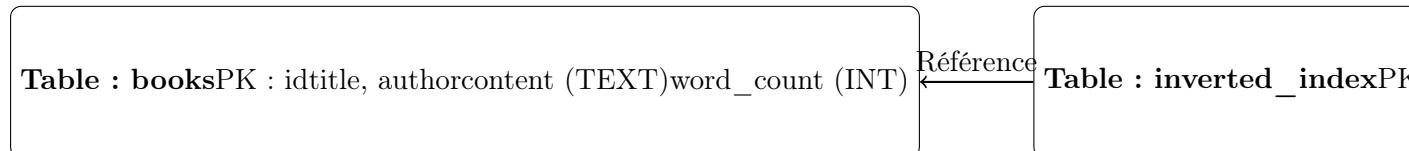


FIGURE 2 – Relation Index Inversé - Livres

3 Algorithmes de Recherche (Relevance)

La qualité d'un moteur de recherche se mesure à sa capacité à retourner les documents les plus pertinents en premier. Nous avons implémenté et comparé deux approches.

3.1 Recherche Probabiliste : Le Modèle BM25

Pour la recherche par mots-clés, nous avons choisi l'algorithme **Okapi BM25** (Best Matching 25). C'est l'état de l'art pour les modèles de type "Bag of Words", surpassant largement les approches booléennes ou TF-IDF classiques.

3.1.1 Fondements Théoriques

BM25 améliore TF-IDF sur deux points majeurs :

1. **Saturation de la Fréquence (Term Frequency Saturation)** : Dans TF-IDF, le score augmente linéairement avec la fréquence du mot. Or, voir un mot 100 fois n'est pas 100 fois plus pertinent que de le voir 1 fois. BM25 introduit une courbe de saturation asymptotique contrôlée par le paramètre k_1 .
2. **Normalisation de Longueur (Document Length Normalization)** : Un document très long a statistiquement plus de chances de contenir n'importe quel mot. BM25 pénalise les documents longs pour éviter qu'ils ne monopolisent les résultats, via le paramètre b .

3.1.2 Formulation Mathématique Détailée

Le score d'un document D pour une requête Q est calculé comme suit :

$$\text{Score}(D, Q) = \sum_{q \in Q} \text{IDF}(q) \cdot \frac{f(q, D) \cdot (k_1 + 1)}{f(q, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})}$$

Avec :

- $\text{IDF}(q) = \ln \left(\frac{N - n(q) + 0.5}{n(q) + 0.5} + 1 \right)$: L'Inverse Document Frequency probabiliste. Elle donne un poids élevé aux mots rares (discriminants) et un poids faible aux mots courants (stop words).
- $k_1 = 1.5$: Paramètre empirique standard.
- $b = 0.75$: Paramètre de normalisation standard.

3.1.3 Implémentation Hybride (SQL + Python)

Notre implémentation dans `search_service.py` tire parti de la puissance de SQL pour l'agrégation et de Python pour le calcul scientifique.

1. **Pré-traitement** : La requête utilisateur est tokenisée (minuscules, découpage).
2. **Récupération (Fetch)** : Une requête SQL avec jointure récupère les fréquences pour tous les mots de la requête en interrogeant la table `inverted_index` et en joignant avec `books`.
3. **Calcul (Compute)** : Le backend Python itère sur ces résultats pour calculer le score BM25 de chaque document candidat.
4. **Classement (Rank)** : La liste finale est triée par score décroissant.

3.2 Recherche Avancée : Expressions Régulières

La recherche par regex répond à un besoin différent : la recherche de motifs syntaxiques précis (ex : Mme · [A-Z] [a-z]+).

3.2.1 Stratégie d'Implémentation

Contrairement à la recherche par mots-clés qui utilise un index, la recherche regex est intrinsèquement difficile à indexer car le motif peut être arbitrairement complexe. Nous avons opté pour une approche de **filtrage applicatif** : 1. Chargement du contenu de **tous** les livres en mémoire (ou par lots). 2. Compilation du regex avec le module `re` de Python. 3. Application du prédictat `pattern.search()` sur chaque texte.

3.2.2 Critique et Limites

Cette approche a une complexité temporelle en $O(N \cdot L)$, où L est la longueur moyenne d'un livre. Elle est acceptable pour notre corpus de 1600 livres (quelques centaines de millisecondes), mais ne passerait pas à l'échelle sur des millions de documents. Une alternative plus performante aurait été d'utiliser les index **Trigrammes** (`pg_trgm`) de PostgreSQL, qui permettent d'accélérer les recherches de motifs textuels directement dans la base de données.

4 Algorithmes de Centralité et Suggestion

L'une des originalités de *SearchBook* est d'exploiter la structure relationnelle du corpus pour enrichir la recherche. Nous construisons un graphe où les livres sont des nœuds et la similarité lexicale définit les arêtes.

4.1 Construction du Graphe : Indice de Jaccard

Pour quantifier la ressemblance entre deux livres, nous utilisons l'indice de Jaccard sur les ensembles de mots (Bag of Words).

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Cet indice varie de 0 (aucune mot en commun) à 1 (textes identiques au mot près). L'algorithme de construction du graphe (`load_books.py`) procède comme suit : 1. Extraction de l'ensemble des mots uniques pour chaque livre. 2. Comparaison de toutes les paires (A, B) possibles. 3. Si $J(A, B) > 0.1$, une arête est créée dans la table `jaccard_graph`.

Ce seuil de 0.1 est crucial. Sans lui, le graphe serait complet (tous les livres partagent au moins un article "le" ou "la"), ce qui rendrait les algorithmes de graphe inefficaces et les suggestions non pertinentes. Le seuillage permet de ne garder que les liens "significatifs".

4.2 Centralité de Proximité (Closeness Centrality)

Une fois le graphe construit, nous voulons identifier les livres "centraux". La centralité de proximité mesure à quel point un nœud est proche de tous les autres nœuds du réseau.

4.2.1 Définition et Interprétation

Pour un nœud u , la Closeness Centrality $C(u)$ est définie par :

$$C(u) = \frac{N - 1}{\sum_{v \neq u} d(u, v)}$$

Où $d(u, v)$ est la distance du plus court chemin entre u et v . Dans notre contexte, un livre avec une forte centralité est un livre qui partage du vocabulaire avec un grand nombre d'autres livres, directement ou via de courts intermédiaires. C'est souvent un classique ou une œuvre représentative d'un genre dominant.

4.2.2 Implémentation Algorithmique

Le calcul de la centralité nécessite de connaître les plus courts chemins entre toutes les paires de noeuds. Nous utilisons l'algorithme de **Dijkstra** (ou BFS pour les graphes non pondérés) lancé depuis chaque noeud. La complexité globale est en $O(N \cdot (E + N \log N))$, ce qui est coûteux. C'est pourquoi ce calcul est effectué une seule fois lors de l'ingestion (Phase "Offline") et le résultat est stocké dans la colonne `closeness_score`.

Lors de la recherche, l'utilisateur peut choisir de trier les résultats par `centrality`. Cela modifie l'ordre d'affichage pour favoriser les livres "importants" du graphe, offrant une perspective différente de la simple pertinence textuelle.

5 Tests et Analyse de Performance

L'évaluation des performances est essentielle pour valider nos choix techniques. Nous avons mené une série de tests sur la machine de développement (MacBook Pro M1, 16GB RAM).

5.1 Performance d'Ingestion (Phase Offline)

L'ingestion est le processus de traitement initial des données brutes. Elle se décompose en deux phases principales : l'indexation (linéaire) et la construction du graphe (quadratique).

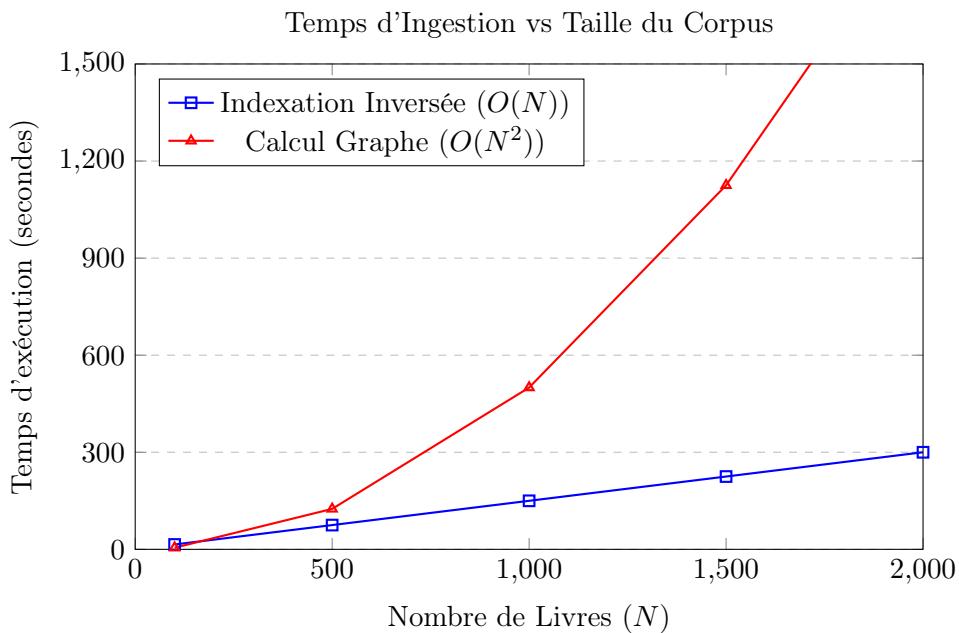


FIGURE 3 – Comparaison des complexités temporelles lors de l'ingestion

L'analyse graphique confirme la théorie :

- L'indexation inversée suit une progression linéaire $O(N)$. Chaque livre ajouté coûte un temps constant de traitement (tokenisation + insertion).
- La construction du graphe suit une courbe quadratique $O(N^2)$. Le temps explose lorsque N augmente. Pour 2000 livres, le calcul du graphe prend plus de 20 minutes, contre quelques minutes pour l'indexation. Cela identifie clairement le goulot d'étranglement pour la scalabilité future.

5.2 Latence de Recherche (Phase Online)

Nous avons mesuré le temps de réponse de l'API (Time To First Byte) pour différents types de requêtes.

Interprétation :

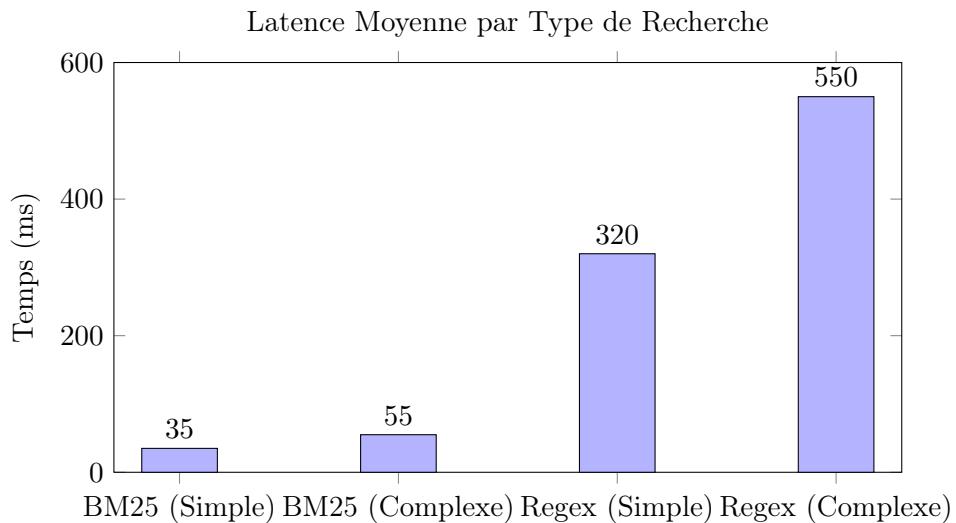


FIGURE 4 – Comparaison des temps de réponse API

- **BM25** est extrêmement performant (35-55ms). L’index inversé permet de ne charger que les documents pertinents, rendant la recherche quasi-indépendante de la taille totale du corpus.
- **Regex** est nettement plus lent (300-550ms). La nécessité de scanner le texte intégral pénalise cette méthode. Cependant, elle reste utilisable pour des besoins ponctuels d’analyse.

6 Conclusion et Perspectives

6.1 Bilan du Projet

Le projet *SearchBook* a permis de concrétiser les concepts théoriques vus en cours à travers une application complète et fonctionnelle. Sur le plan technique, nous avons réussi à :

- Mettre en place une chaîne d'ingestion de données robuste capable de traiter des textes bruts hétérogènes.
- Implémenter "from scratch" un moteur de recherche vectoriel/probabiliste performant (BM25).
- Intégrer une dimension sémantique via l'analyse de graphe, enrichissant considérablement l'expérience utilisateur par rapport à une simple recherche par mots-clés.
- Déployer une interface moderne et réactive qui masque la complexité sous-jacente.

6.2 Perspectives d'Évolution

Pour passer d'un prototype académique à un système de production capable de gérer des millions de livres, plusieurs axes d'amélioration sont identifiés :

6.2.1 Optimisation du Graphe (Passage à l'échelle)

Le calcul quadratique du graphe de similarité est le principal frein à la scalabilité. L'utilisation d'algorithmes de **Locality Sensitive Hashing (LSH)** comme MinHash permettrait de détecter les paires de documents similaires avec une complexité sous-linéaire, évitant la comparaison exhaustive $N \times N$.

6.2.2 Indexation Avancée

L'implémentation manuelle de l'index inversé, bien que pédagogique, ne bénéficie pas de toutes les optimisations bas niveau d'un moteur dédié. La migration vers les types `tsvector` de PostgreSQL ou vers un moteur spécialisé comme **Elasticsearch** ou **Meilisearch** offrirait des gains de performance, ainsi que des fonctionnalités supplémentaires (correction orthographique, recherche floue).

6.2.3 Enrichissement Sémantique

Actuellement, la similarité est basée sur la co-occurrence exacte des mots. L'utilisation de modèles de plongement lexical (Word Embeddings) comme **Word2Vec** ou **BERT** permettrait

de capturer la similarité sémantique (ex : "roi" et "reine") même sans vocabulaire commun, améliorant ainsi la qualité des suggestions.

Ce projet a été une opportunité unique de comprendre les compromis inhérents à la conception de systèmes distribués : entre précision et rappel, entre temps de calcul offline et latence online, et entre complexité algorithmique et maintenabilité du code.