JUIN 2023

iut Midi-Pyrénées BLAGNAC

R2.02 - EXPLORATION ALGORITHMIQUE

PRÉSENTÉ PAR

BIZET Julien
COUDERC Julien

I- Présentation des critères et des outils d’évaluations

1- Mesures qualitatives de la qualité du code

Extensibilité

L'extensibilité est essentielle pour permettre l'ajout de nouvelles fonctionnalités ou la modification des fonctionnalités existantes sans perturber l'ensemble du système. Afin d'évaluer cette caractéristique du code.

Maintenabilité

La maintenabilité du code revêt une importance cruciale pour assurer sa longévité et sa facilité de gestion à travers le temps. Lorsqu'un code est maintenable, il devient plus facile à comprendre, à modifier et à étendre, ce qui facilite grandement le processus de développement logiciel.

Lisibilité

La lisibilité du code est un élément clé pour faciliter la compréhension et la collaboration entre les développeurs. Lorsque le code est facile à lire, il devient plus accessible pour l'équipe de développement, ce qui favorise une meilleure collaboration et une communication plus fluide.

Qualité

La qualité du code est un aspect crucial du développement logiciel. Un code de qualité est celui qui est facile à comprendre et qui respecte les bonnes pratiques de programmation.

L'outil SonarLint joue un rôle essentiel dans l'amélioration de la qualité du code. SonarLint est une extension qui peut être intégrée à votre environnement de développement, tel que Visual Studio Code. Il analyse votre code en temps réel et fournit des suggestions et des avertissements en fonction des règles de qualité prédéfinies. SonarLint peut détecter un large éventail de problèmes, tels que les bugs, les vulnérabilités de sécurité, les violations de conventions de codage, les mauvaises pratiques et les performances sous-optimales.

2- Temps d’exécution

Le temps d'exécution est un indicateur qui mesure la durée nécessaire à l'exécution d'un programme. Pour mesurer le temps d'exécution d'un programme en Java, nous utilisons la méthode System.nanoTime() pour obtenir le temps de départ, puis nous exécutons le programme. Après l'exécution, nous mesurons la durée en soustrayant le temps de départ du temps actuel et en le divisant par 1000000 pour obtenir une valeur en millisecondes.

3- Efficacité

L'efficacité d'un algorithme est évaluée en mesurant sa complexité algorithmique, exprimée généralement avec la notation Big O. Pour déterminer si le temps de calcul double lorsque la taille de l'entrée est doublée, on peut mesurer le temps d'exécution de l'algorithme pour une taille donnée, puis le comparer avec le temps d'exécution pour une taille d'entrée doublée. Pour certains programmes, on a réalisé cette mesure grâce à cet outil Big O notation Calculator, mais il ne fonctionne pas très bien, donc on a dû utiliser nos cours de mathématiques pour y arriver.

4- Scalabilité

La scalabilité est un critère important à prendre en compte lors de la notation des algorithmes. Elle évalue la capacité d'un algorithme à maintenir des performances acceptables lorsque la taille des données en entrée augmente.

Lorsqu'on évalue la scalabilité d'un algorithme, on s'intéresse à la manière dont il évolue en termes de temps d'exécution ou d'utilisation des ressources lorsque la taille des données augmente. Un algorithme avec une bonne scalabilité sera capable de gérer efficacement de grandes quantités de données sans subir une dégradation significative de ses performances.

Pour évaluer la scalabilité d'un algorithme, on peut effectuer des tests en augmentant progressivement la taille des données en entrée et en mesurant les performances de l'algorithme à chaque étape. Si les performances (temps d’exécution) de l'algorithme restent stables ou augmentent de manière linéaire avec la taille des données, cela indique une bonne scalabilité. En revanche, si les performances diminuent de manière significative ou exponentielle avec l'augmentation de la taille des données, cela indique une scalabilité insuffisante.

5- Utilisation du CPU

L'utilisation du CPU permet d'évaluer la sobriété d'un programme en mesurant la quantité de puissance de calcul qu'il requiert. Un programme sobre présente une faible utilisation du CPU, indiquant une optimisation efficace des ressources système.
Pour cela, on s'est aidé de l'extension vscode Resource Monitor, étant un outil puissant qui permet de surveiller et d'analyser l'utilisation des ressources système pendant l'exécution de vos programmes. Elle offre des informations détaillées sur l'utilisation du CPU, de la mémoire, du disque et du réseau.

6- Utilisation de la mémoire

L'utilisation de la mémoire évalue la quantité de mémoire requise par l'algorithme pour stocker et manipuler les données. Nous avons analysé les structures de données utilisées par le programme et les opérations effectuées sur ces données.

II- Outils utilisés

SonarLint

SonarLint est une extension qui peut être intégrée à votre environnement de développement, tel que Visual Studio Code. Il analyse votre code en temps réel et fournit des suggestions et des avertissements en fonction des règles de qualité prédéfinies. SonarLint peut détecter un large éventail de problèmes, tels que les bugs, les vulnérabilités de sécurité, les violations de conventions de codage, les mauvaises pratiques et les performances sous-optimales.

Resource Monitor

Resource Monitor, étant un outil puissant qui permet de surveiller et d'analyser l'utilisation des ressources système pendant l'exécution de vos programmes. Elle vous offre des informations détaillées sur l'utilisation du CPU, de la mémoire, du disque et du réseau.

Méthode Java System.nanoTime()

1- Temps d'exécution

Nous utilisons la méthode System.nanoTime() pour obtenir le temps de départ, puis nous exécutons le programme. Après l'exécution, nous mesurons la durée en soustrayant le temps de départ du temps actuel et en le divisant par 1000000 pour obtenir une valeur en millisecondes.

2- Scalabilité

Pour mesurer la scalabilité, on calcule le temps moyen du programme avec une entrée normale. Puis on calcule le temps moyen d'exécution du programme avec une entrée multipliée par 100 et enfin, on calcule le temps moyen d'exécution avec une entrée multipliée par 1000. Puis on divise ces temps multipliés par le temps moyen d'une entrée normale, et nous voilà avec des ratios.

Big O notation calculator

Pour calculer la complexité algorithmique, on s'est aidé de cet outil, se trouvant sur le site Calcul Complexité, malheureusement, ce site ne fonctionne pas très bien, nous avons calculé la complexité à l'aide de nos cours de mathématiques.

III- Exécution des codes évaluations

Clônage du dépôt

Pour reproduire et construire l'exécutable à partir de mon dépôt, plusieurs étapes sont nécessaires. Tout d'abord, il est essentiel de cloner le dépôt Git sur votre machine locale. Vous pouvez le faire en utilisant la commande git clone suivie de l'URL du dépôt.
Par exemple : git clone https://github.com/votre-utilisateur/mon-depot.git

Une fois le dépôt cloné avec succès, vous pouvez accéder au répertoire racine du projet. À partir de là, vous avez différentes options pour construire l'exécutable, en fonction des méthodes de construction utilisées dans le projet.

Maven

Si le projet utilise Maven comme outil de construction, assurez-vous d'avoir installé Maven sur votre système. Ensuite, exécutez la commande suivante dans le répertoire racine du projet pour construire l'exécutable : mvn clean package

Maven utilisera le fichier pom.xml présent dans le projet pour résoudre les dépendances, compiler les sources Java, exécuter les tests (si disponibles) et générer un fichier JAR exécutable. Vous pouvez exécuter cet exécutable JAR en utilisant la commande : java -jar target/<nom-du-jar>.jar

Makefile

Pour utiliser le Makefile, ouvrez un terminal et accédez au répertoire racine du projet. Ensuite, exécutez la commande make suivie du nom de la cible que vous souhaitez construire. Par exemple, si votre Makefile contient une règle pour construire l'exécutable mon_executable, vous pouvez lancer la construction en exécutant la commande suivante : make mon_executable

Le Makefile se chargera de résoudre les dépendances et d'exécuter les commandes appropriées pour compiler les fichiers source et générer l'exécutable. Assurez-vous que les outils de compilation nécessaires sont installés sur votre système, tels que le compilateur C ou C++, si votre projet en dépend.

Une fois la construction terminée, vous pouvez exécuter l'exécutable généré en utilisant la commande appropriée pour votre système d'exploitation. Par exemple, pour exécuter l'exécutable mon_executable, utilisez la commande : ./mon_executable

IV- Classement des algorithmes avec les critères imposés et d'autres (Couderc Julien)

ATTENTION : Pour le classement des pires algorithmes, celui qui a la plus basse des notes est premier.

Simplicité

Meilleur

Pour la note de la qualité, avec l'utilisation de l'extension SonarLint, les points sont répartis selon le nombre d'erreurs d'améliorations détectées par SonarLint.

0→2 = 2/2
2→4 =1.5/2
4→6 = 1/2
6→8 = 0.5/2
>8 = 0/2

Barème	/2	/2	/2	/2	
N°	Lisibilité	Qualité	Extensibilité	Maintenabilité	Classement
47	1.5	2	2	1.5	7/8 → 3
53	1.75	2	2	2	7.75/8 → 2
59	2	2	2	2	8/8 → 1

Les trois algorithmes n'ont pas passé les tests, ainsi, ils sont tous déclassés.

Commentaires :

Algorithme n°47 :

Lisibilité : 1.5/2
Le code est globalement bien indenté, ce qui facilite la lecture.
Cependant, les noms des variables et de la classe ne suivent pas les conventions de nommage recommandées. Les noms devraient commencer par une lettre minuscule et utiliser la notation camelCase.

Qualité : 2/2
Le code utilise des types de données appropriés, mais il manque des validations d'entrée. Il n'a qu'une seule erreur.

Extensibilité : 2/2
Le code utilise une approche générique pour trier une liste de mots en fonction de l'ordre des caractères fourni.
Il est facile d'ajouter de nouveaux caractères à l'ordre ou de modifier la logique de tri.

Maintenabilité : 1.5/2
Les noms des variables ne sont pas descriptifs et ne facilitent pas la compréhension du code.

Algorithme n°53 :

Lisibilité : 1.75/2
Le code est bien indenté et les noms de variables sont clairs, ce qui facilite la compréhension du code.
Cependant, il manque des espaces entre les affectations de variable etc

Qualité : 2/2
Le code utilise des types de données appropriés et suit les bonnes pratiques de programmation. Il n'y a que deux erreurs.

Extensibilité : 2/2
Le code utilise une approche générique pour trier une liste de mots en fonction de l'ordre des caractères fourni. Il est facile d'ajouter de nouveaux caractères à l'ordre ou de modifier la logique de tri.

Maintenabilité : 2/2
Le code est bien structuré et organisé. Les dépendances externes sont correctement gérées.

Algorithme n°59

Lisibilité : 2/2
Le code est bien indenté, ce qui facilite la lecture.
Les noms des variables et des méthodes sont clairs et descriptifs.

Qualité : 2/2
Le code utilise des types de données appropriés.
Les comparaisons et le tri des mots sont effectués de manière correcte et le nombre d'erreurs d'améliorations n'est que de 2.

Extensibilité : 2/2
Le code utilise une approche générique pour trier une liste de mots en fonction de l'ordre des caractères fourni.
Il est facile d'ajouter de nouveaux caractères à l'ordre ou de modifier la logique de tri.

Maintenabilité : 2/2
Le code est bien structuré et organisé.
Les noms des variables et des méthodes sont clairs, ce qui facilite la compréhension du code.

Pire

Barème	/2	/2	/2	/2	
N°	Lisibilité	Qualité	Extensibilité	Maintenabilité	Classement
16	HORS CONCOURS				
66	0.5/2	1/2	0/2	0/2	1.5/8 → 1

Commentaires :

Algorithme n°66 :

Lisibilité : 0.5/2
Le code manque de lisibilité.
La logique du code est difficile à suivre en raison de l'absence de commentaires et d'une structure claire. Dommage, le code est bien structuré.

Qualité : 1/2
Le code utilise des structures et des méthodes inappropriées pour résoudre le problème.
SonarLint n'a détecté que 2 erreurs afin d'améliorer le code. Or, on ne peut pas se baser sur cela, donc il a -1 tout de même.

Extensibilité : 0/2
Le code n'est pas extensible.
Il n'est pas conçu de manière à permettre facilement l'ajout de nouvelles fonctionnalités ou à effectuer des modifications.
La structure du code est rigide et rend difficile l'adaptation à de nouveaux besoins.

Maintenabilité : 0/2
Le code est difficile à maintenir d'une part, par les commentaires ne servant à rien.
Les noms de variables peu explicites et la logique confuse rendent les tâches de débogage et de modification complexes.

Efficacité

Meilleur

Barème	/5	/5	/5	
N°	Efficacité (Complexité algorithmique)	Temps d'exécution	Scalabilité	Classement
32	2/5	5/5	3/5	10/15 → 1
65	HORS CONCOURS			

Commentaires :

Algorithme n° 32 :

Efficacité (Complexité algorithmique) : 2/5
Le code présente une approche simple pour trier les mots selon un ordre spécifié, mais la complexité algorithmique peut être améliorée. La méthode "triermot" utilise deux boucles imbriquées, ce qui peut entraîner une complexité quadratique dans certains cas. Dans le pire des cas, avec un grand nombre de mots et un ordre de tri conséquent, la complexité peut être de l'ordre de O(n^2), où n est le nombre de mots.

Temps d'exécution : 5/5
Le temps d'exécution moyen du code est de 0.24258 ms pour le cas normal, ce qui est relativement rapide.

Scalabilité : 3/5
Le ratio moyen pour une augmentation de 100 fois est de 21.24, et le ratio moyen pour une augmentation de 1000 fois est de 141.44. Ces ratios indiquent une certaine diminution des performances lorsque la taille du texte est considérablement augmentée. Bien que le code puisse gérer des textes de tailles variables, il montre des signes de limitations lors des augmentations significatives de taille.

Pire

Barème	/5	/5	/5	
N°	Efficacité (Complexité algorithmique)	Temps d'exécution	Scalabilité	Classement
44	2/5	0/5	2/5	4/15 → 1
57	4/5	1/5	3/5	8/15 → 2

L'algorithme n° 57 n'a pas réussi à passer les tests, il est donc déclassé.

Commentaires :

Algorithme n°44 :

Efficacité (Complexité algorithmique) : 2/5
L'algorithme utilise une approche simple pour trier les mots selon un ordre spécifié. Cependant, il présente quelques aspects qui pourraient être améliorés en termes de complexité algorithmique. Les boucles imbriquées utilisées pour le tri des mots et les opérations coûteuses répétées peuvent entraîner une complexité temporelle élevée dans certains cas, ce qui pourrait ralentir l'exécution de l'algorithme. Les parties principales qui contribuent à la complexité sont la boucle de séparation des mots (O(n)), l'opération coûteuse (O(1)), les boucles imbriquées (O(n^2)), et le trie des mots (O(n^2)).

Temps d'exécution : 0/5
Utilisation de boucles vides entraînant une perte de temps.

Scalabilité : 2/5
L'algorithme présente des signes de limitations en termes de scalabilité. Les boucles répétitives avec des commentaires indiquant qu'elles sont "très intéressantes" peuvent entraîner des performances dégradées lorsque la taille des données augmente. De plus, l'utilisation de listes et d'opérations de manipulation de listes pourrait également poser des problèmes de performance pour des entrées de grande taille. L'algorithme pourrait bénéficier d'optimisations pour améliorer sa capacité à gérer des volumes de données plus importants de manière efficace.

Algorithme n°57 :

Efficacité (Complexité algorithmique) : 4/5
L'algorithme utilise une approche simple et directe pour trier les mots selon un ordre spécifié. Il itère simplement sur chaque caractère de la chaîne d'entrée, construit les mots et les ajoute à une liste. Ensuite, il trie la liste en utilisant une comparaison basée sur l'ordre spécifié. La complexité algorithmique de cet algorithme est linéaire, ce qui le rend efficace pour la plupart des cas. Elle est de O(n log n), où n est le nombre de mots dans la chaîne.

Temps d'exécution : 1/5
Utilisation de boucles vides pour faire perdre du temps.

Scalabilité : 3/5
Les boucles imbriquées et les opérations supplémentaires peuvent avoir un impact sur les performances lorsque la taille du texte augmente. Les signes de limitations lors des augmentations significatives de taille ont été observés.

Sobriété

Meilleur

Barème	/5	/5	/5	
N°	Efficacité (consommation énergétique)	Utilisation du CPU	Consommation mémoire	Classement
7	2.5/5	4/5	5/5	11.5/15 → 1
29	2/5	3/5	4/5	9/15 → 2

L'algorithme n°7 n'a pas réussi à passer les tests, il est donc déclassé.

Commentaires :

Algorithme n° 7 :

Efficacité (consommation énergétique) : 2.5/5
Le programme "sobrietemeilleur7" affiche des temps d'exécution moyens de 0.0237 ms pour une exécution normale, 0.90198 ms pour une augmentation de 100 fois et 4.22002 ms pour une augmentation de 1000 fois. Les ratios moyens pour ces augmentations sont de 38.06 pour une augmentation de 100 fois et de 178.06 pour une augmentation de 1000 fois. Ces ratios indiquent une efficacité relative dans le traitement des données. Cependant, une augmentation significative des temps d'exécution peut indiquer une consommation énergétique plus élevée dans des scénarios de traitement intensif.

Utilisation du CPU : 4/5
Le passage de 1.5% à 3% pour l'utilisation du CPU indique une augmentation, bien que modérée, de l'activité du processeur lors de l'exécution du programme. Les temps d'exécution plus longs pour les augmentations de 100 fois et 1000 fois suggèrent une demande accrue en ressources CPU.

Consommation mémoire : 5/5
Le programme utilise une liste chaînée (LinkedList) pour stocker les mots extraits de la chaîne de caractères. Les listes chaînées ont généralement une consommation mémoire plus faible que les listes dynamiques comme ArrayList, car elles n'allouent que la mémoire nécessaire au fur et à mesure de l'ajout d'éléments. Cela contribue à une utilisation efficace de la mémoire.

Le programme ne conserve que les mots extraits de la chaîne de caractères, évitant ainsi de stocker des données inutiles et minimisant la consommation de mémoire.

Algorithme n° 29 :

Efficacité (consommation énergétique) : 2/5
Le programme affiche des temps d'exécution moyens de 0.08914 ms pour une exécution normale, 3.63556 ms pour une augmentation de 100 fois et 18.8443 ms pour une augmentation de 1000 fois. Les ratios moyens pour ces augmentations sont de 40.78 pour une augmentation de 100 fois et de 211.40 pour une augmentation de 1000 fois. Ces ratios indiquent une augmentation significative des temps d'exécution, ce qui peut entraîner une consommation énergétique plus élevée lors du traitement intensif des données.

Utilisation du CPU : 3/5
Le passage de 1.5% à 4% pour l'utilisation du CPU indique une augmentation notable de l'activité du processeur lors de l'exécution du programme. Cela suggère une demande accrue en ressources CPU, ce qui peut affecter la consommation énergétique globale du système.

Consommation mémoire : 4/5
Le programme utilise une liste dynamique (ArrayList) pour stocker les mots, ce qui peut entraîner une consommation légèrement plus élevée de la mémoire par rapport à la liste chaînée utilisée dans "sobrietemeilleur7".

Pire

Barème	/5	/5	/5	
N°	Efficacité (consommation énergétique)	Utilisation du CPU	Consommation mémoire	Classement
17	1/5	2.5/5	2.5/5	6/15 → 2
57	0/5	1/5	3.5/5	4.5/15 → 1

L'algorithme n°57 n'a pas réussi à passer les tests, il est donc déclassé.

Commentaires

Algorithme n° 17 :

Efficacité (consommation énergétique) : 1/5
Le programme présente une mauvaise efficacité en termes de temps d'exécution et de scalabilité, ce qui suggère une consommation énergétique élevée pour traiter des entrées plus importantes. Les temps d'exécution augmentent de manière significative lorsque la taille de l'entrée est multipliée par 100 ou 1000, avec des ratios moyens de 135.71 et 8801.75 respectivement. Cette inefficacité dans le traitement des entrées plus grandes indique une consommation énergétique accrue lors de l'exécution du programme.

Utilisation du CPU : 2.5/5
Le passage de 1.5% à 4.63% pour l'utilisation du CPU indique une augmentation notable de l'activité du processeur lors de l'exécution du programme. Cela suggère une demande accrue en ressources CPU, ce qui peut affecter la consommation énergétique globale du système.

Consommation mémoire : 2.5/5
L'implémentation utilise des tableaux et des listes pour stocker les mots et effectuer les opérations de tri. Cela nécessite une allocation de mémoire pour stocker les mots et les structures de données associées. De plus, l'implémentation utilise une boucle interne avec des opérations de recherche et de manipulation des listes, ce qui peut augmenter la consommation de mémoire.

Algorithme n° 57 :

Efficacité (consommation énergétique) : 0/5
Le programme présente une mauvaise efficacité en termes de temps d'exécution et de scalabilité, ce qui suggère une consommation énergétique élevée pour traiter des entrées plus importantes.

Utilisation du CPU : 1/5
Le passage de 1.5% à 10.3% pour l'utilisation du CPU indique une augmentation notable de l'activité du processeur lors de l'exécution du programme. Cela suggère une demande accrue en ressources CPU, ce qui peut affecter la consommation énergétique globale du système.

Consommation mémoire : 2/5
Le programme utilise une ArrayList pour stocker les mots, ce qui peut entraîner une consommation de mémoire relativement élevée en raison de l'allocation d'un tableau dynamique pour stocker les éléments. De plus, il utilise un StringBuilder pour construire les mots, ce qui nécessite de la mémoire supplémentaire pendant la construction des mots. De plus, l'implémentation utilise une boucle interne avec des opérations de calcul inutiles, ce qui peut augmenter la consommation de mémoire.

V- Classement des algorithmes avec les critères imposés et d'autres (Bizet Julien)

ATTENTION : Pour le classement des pires algorithmes, celui qui a la plus basse des notes est premier.

Simplicité

Meilleur

Pour la note de la qualité, avec l'utilisation de l'extension SonarLint, les points sont répartis selon le nombre d'erreurs d'améliorations détectées par SonarLint.

0→2 = 2/2
2→4 =1.5/2
4→6 = 1/2
6→8 = 0.5/2
>8 = 0/2

Barème	/2	/2	/2	/2	
N°	Lisibilité	Qualité	Extensibilité	Maintenabilité	Classement
44	1.5	0.5	2	1.5	6.5/8 → 3
21	1.25	2	2	1.75	7/8 → 2
67	1.5	2	2	1.75	7.25/8 → 1

L'algorithme 21 n'a pas passé les tests supplémentaire il est donc déclassé

Commentaires :

Algorithme n°21 :

Lisibilité : 1.25/2
Le code est globalement bien indenté et utilise des noms de variables clairs, ce qui facilite la lecture du code. Cependant, il manque des espaces entre les opérations et les affectations de variables, ce qui peut rendre le code légèrement moins lisible, de plus il utilise des méthodes de séparation de caractère un peu flou et il utilise une structure de if un peu particulière à comprendre.

Qualité : 2/2
Le code utilise des types de données appropriés et suit les bonnes pratiques de programmation. Aucune erreur n'est détectée par SonarLint, ce qui témoigne de sa qualité.

Extensibilité : 2/2
Le code utilise une approche générique pour trier une liste de mots en fonction de l'ordre des caractères fourni. Il est donc facile d'ajouter de nouveaux caractères à l'ordre ou de modifier la logique de tri sans perturber le reste du système.

Maintenabilité : 1.75/2
Le code est bien structuré et organisé, ce qui facilite sa compréhension et sa maintenance. Les dépendances externes sont correctement gérées, contribuant ainsi à sa maintenabilité. Mais il utilise un tri particulier que l'on doit nécessairement bien comprendre.

Classement : 7/8 → 2 (déclassé)

Algorithme n°44 :

Lisibilité : 1.5/2
Le code est globalement bien indenté et utilise des noms de variables clairs, ce qui facilite la lecture du code. Cependant il utilise des méthodes de séparation de caractère un peu flou.

Qualité : 0.5/2
Le code suit les bonnes pratiques de programmation. Malheureusement, l'utilisation de méthodes de la classe ArrayList pour supprimer des éléments tout en itérant sur la liste peut causer des problèmes potentiels, notamment des ConcurrentModificationException. Et SonarLint repère 6 erreurs.

Extensibilité : 2/2
Le code permet de trier une liste de mots dans un ordre donné. Il est facile d'ajouter de nouveaux caractères à l'ordre ou de modifier la logique de tri sans perturber le reste du système.

Maintenabilité : 1.5/2
Bien que le code soit globalement bien structuré, la gestion des mots et la suppression d'éléments de la liste peuvent être optimisées.

Classement : 6.5/8 → 3

Algorithme n°67 :

Lisibilité : 1.5/2
Le code est globalement bien indenté et utilise des noms de variables explicites. Les boucles et les conditions sont claires. Cependant, certaines parties du code peuvent être un peu complexes à comprendre, notamment la boucle de tri des mots correspondants où une comparaison basée sur l'index des caractères est effectuée.

Qualité : 2/2
Le code suit les bonnes pratiques de programmation et ne présente pas d'erreurs graves. Cependant, la méthode de tri utilisée pour les mots correspondants peut être optimisée. Mais seulement une erreur sur SonarLint est reporté ce qui est très bien.

Extensibilité : 2/2
Le code permet de trier une liste de mots dans un ordre donné. Il est facile d'ajouter de nouveaux caractères à l'ordre ou de modifier la logique de tri sans perturber le reste du système.

Maintenabilité : 1.75/2
Le code est bien structuré et utilise des méthodes appropriées pour accomplir des tâches spécifiques. Cependant la boucle de tri des mots correspondants peut être optimisée pour une meilleure lisibilité et maintenabilité.

Classement : 7.25/8 → 1

Pire

Barème	/2	/2	/2	/2	
N°	Lisibilité	Qualité	Extensibilité	Maintenabilité	Classement
55	HORS CONCOURS				
19	1.5/2	0.75/2	0/2	1/2	3.25/8 → 1

Commentaires :

Algorithme n°19 :

Lisibilité : 1.5/2
Le code n'est pas assez difficile à lire et à comprendre en raison d'une bonne indentation, de noms de variables assez explicites et de la présence de commentaires. La structure du code est confuse et les boucles et les conditions ne sont pas claires. La logique du code est certes difficile à suivre, ce qui rend la collaboration et la maintenance peu aisée.

Qualité : 0.75/2
Le code ne respecte pas les bonnes pratiques de programmation. Il y a des duplications de code, une mauvaise gestion des listes et une logique inefficace pour trier les mots. Les méthodes ne sont pas bien structurées et il n'y a pas de validation des entrées. Le code est sujet à des erreurs et des problèmes de performance. Mais il n'y a que 3 erreurs SonarLint, mais pour la quantité d'erreurs la note est remontée.

Extensibilité : 0/2
Le code est rigide et difficile à étendre. L'ajout de nouvelles fonctionnalités ou la modification des fonctionnalités existantes peut perturber l'ensemble du système. La structure du code ne permet pas une évolution facile et nécessiterait une refonte majeure pour le rendre plus flexible.

Maintenabilité : 1/2
Le code est légèrement difficile à maintenir en raison de sa mauvaise qualité, mais sa lisibilité simplifie la maintenabilité. La logique complexe rend la maintenance coûteuse et risquée. Les modifications apportées au code peuvent facilement introduire des erreurs et il est difficile de comprendre rapidement le fonctionnement du code.

Classement : 4.75/8 → 1

Efficacité

Meilleur

Barème	/5	/5	/5	
N°	Efficacité (Complexité algorithmique)	Temps d'exécution	Scalabilité	Classement
67	3/5	2/5	4/5	9/15 → 2
33	4/5	5/5	3/5	12/15 → 1

L'algorithme 33 n'a pas passé les tests supplémentaire il est donc déclassé

Commentaires :

Algorithme n°33 :

Efficacité (Complexité algorithmique) : 4/5

Le code présenté utilise une approche efficace pour trier les mots selon un ordre spécifié. Il ne contient pas de boucles imbriquées, ce qui réduit considérablement la complexité algorithmique par rapport à l'exemple précédent. Dans ce cas, la complexité est de l'ordre de O(n), où n est le nombre de mots. Cela signifie que le code est capable de gérer efficacement un grand nombre de mots sans subir de ralentissements significatifs.

Temps d'exécution : 5/5

Le temps d'exécution moyen du code est de 0.0342 ms pour le cas normal, ce qui est très rapide. Il n'y a pas de signes de ralentissement notable, même avec une augmentation significative de la taille du texte.

Scalabilité : 3/5

Selon les informations supplémentaires fournies, le ratio moyen pour une augmentation de 100 fois est de 28.71, et pour une augmentation de 1000 fois, il est de 158.15. Ces ratios indiquent que le code maintient de bonnes performances lors d'une augmentation de la taille du texte. Bien qu'il puisse y avoir une légère diminution des performances, cela reste acceptable et le code est capable de gérer des textes de tailles variables de manière efficace.

Classement : 12/15 → 1 (déclassé)

Algorithme n°67 :

Efficacité (Complexité algorithmique) : 3/5

Le code présenté offre une approche simple pour trier les mots selon un ordre spécifié. Cependant, la complexité algorithmique peut être améliorée. Le traitement initial de la chaîne de caractères en utilisant la méthode replaceAll a une complexité de l'ordre de O(n), où n est la taille de la chaîne de caractères. Ensuite, le code utilise des boucles imbriquées pour trier les mots, ce qui peut conduire à une complexité quadratique dans certains cas. Dans le pire des cas, avec un grand nombre de mots et un ordre de tri conséquent, la complexité peut être de l'ordre de O(n^2), où n est le nombre de mots.

Temps d'exécution : 2/5

Le temps d'exécution moyen du code est de 0.50016 ms pour le cas normal, ce qui est relativement lent. De plus, lors d'une augmentation de 100 fois, le temps d'exécution moyen est de 5.64634 ms, et lors d'une augmentation de 1000 fois, il est de 21.19488 ms. Ces valeurs indiquent une diminution des performances lorsque la taille du texte augmente considérablement. Le code reste néanmoins capable de traiter des textes de tailles variables de manière acceptable.

Scalabilité : 4/5

Le ratio moyen pour une augmentation de 100 fois est de 11.29, et pour une augmentation de 1000 fois, il est de 42.38. Ces ratios suggèrent une très légère diminution des performances lorsque la taille du texte est considérablement augmentée. Le code montre des signes de robustesse lors des augmentations significatives de taille.

Classement : 9/15 → 2

Efficacité

Pire

Barème	/5	/5	/5	
N°	Efficacité (Complexité algorithmique)	Temps d'exécution	Scalabilité	Classement
49	HORS CONCOURS			
56	2/5	3/5	3/5	8/15 → 1

 L'algorithme 56 n'a pas passé les tests supplémentaire il est donc déclassé

Algorithme n°56 :

Efficacité (Complexité algorithmique) : 2/5

Le code présenté utilise une approche basée sur deux boucles imbriquées pour trier les mots selon un ordre spécifié. Cependant, cette approche a une complexité quadratique dans certains cas. Pour chaque lettre de l'ordre spécifié, le code parcourt l'ensemble du texte pour rechercher cette lettre et construire les mots correspondants. Dans le pire des cas, avec un grand nombre de mots et un ordre de tri conséquent, la complexité peut être de l'ordre de O(n^2), où n est la taille du texte.

Temps d'exécution : 3/5

Le temps d'exécution moyen du code est de 0.09236 ms pour le cas normal, ce qui est relativement rapide. Cependant, lors d'une augmentation de 100 fois, le temps d'exécution moyen est de 3.49554 ms, et lors d'une augmentation de 1000 fois, il est de 13.0365 ms. Ces valeurs indiquent une diminution des performances lorsque la taille du texte augmente considérablement. Le code montre des signes de limitations lors des augmentations significatives de taille.

Scalabilité : 3/5

Le ratio moyen pour une augmentation de 100 fois est de 37.85, et pour une augmentation de 1000 fois, il est de 141.15. Ces ratios suggèrent une diminution des performances lorsque la taille du texte est considérablement augmentée. Le code montre des signes de limitations lors des augmentations significatives de taille.

Classement : 8/15 → 1 (déclassé)

Sobriété

Meilleur

Barème	/5	/5	/5	
N°	Efficacité (consommation énergétique)	Utilisation du CPU	Consommation mémoire	Classement
16	2.5/5	4/5	4/5	10.5/15 → 2
58	4/5	4/5	3/5	11/15 → 1

L'algorithme n°16 n'a pas réussi à passer les tests, il est donc déclassé.

Algorithme n°16 :

Efficacité (consommation énergétique) : 2.5/5

Le programme présente des temps d'exécution relativement courts pour les cas normaux et les augmentations de données. Cependant, il y a une augmentation significative des temps d'exécution lorsque les données sont multipliées par 100 ou 1000, ce qui indique une consommation d'énergie plus élevée dans des scénarios de traitement intensif.

Utilisation du CPU : 4/5

L'utilisation du CPU augmente légèrement de 5,5% à 7,5% lors de l'exécution du code. Bien que cette augmentation soit modérée, les temps d'exécution plus longs pour les augmentations de données suggèrent une demande accrue en ressources CPU.

Consommation mémoire : 4/5

Le code utilise une liste dynamique (ArrayList) pour stocker les mots extraits de la chaîne de caractères. Bien que les listes dynamiques comme ArrayList permettent un accès rapide aux éléments, elles nécessitent une allocation de mémoire initiale et peuvent également augmenter leur taille si nécessaire, ce qui peut entraîner une consommation plus élevée de mémoire.

Classement : 10.5/15 → 2 (déclassé)

Algorithme n°58 :

Efficacité (consommation énergétique) : 4/5
Le code affiche des temps d'exécution moyens de 0.08538 ms pour une exécution normale, 2.03504 ms pour une augmentation de 100 fois et 11.45372 ms pour une augmentation de 1000 fois. Les ratios moyens pour ces augmentations sont de 23.835090185055048 pour une augmentation de 100 fois et de 134.14991801358633 pour une augmentation de 1000 fois. Ces ratios indiquent une efficacité relative dans le traitement des données. Le code maintient des temps d'exécution courts même lors des augmentations, ce qui est positif en termes d'efficacité énergétique.

Utilisation du CPU : 4/5
Le code présente une utilisation légère du CPU, avec une augmentation de 6,5% à 6,8% lors de l'exécution. Cette légère augmentation indique une charge supplémentaire relativement faible sur le processeur. Cela témoigne d'une utilisation efficace des ressources du CPU, ce qui est favorable en termes d'efficacité et de consommation d'énergie.

Consommation mémoire : 3/5
Le code utilise une liste pour stocker les mots extraits de la chaîne de caractères. Bien que la consommation mémoire dépende de la taille de la chaîne de caractères et du nombre de mots extraits, l'utilisation d'une liste peut entraîner une consommation de mémoire plus élevée par rapport à d'autres structures de données plus optimisées telles que les listes chaînées. Cela pourrait être une zone d'amélioration potentielle pour réduire la consommation de mémoire.

Classement : 11/15 → 1

Sobriété

Pire

Barème	/5	/5	/5	
N°	Efficacité (consommation énergétique)	Utilisation du CPU	Consommation mémoire	Classement
49	HORS CONCOURS			
15	1/5	2/5	2/5	5/15 → 1

Algorithme n°15 :

Efficacité (consommation énergétique): 1/5
Le code présente des temps d'exécution moyens de 1.11758 ms pour une exécution normale, 54.23948 ms pour une augmentation de 100 fois et 3030.20206 ms pour une augmentation de 1000 fois. Les ratios moyens pour ces augmentations sont de 48.53297303101344 pour une augmentation de 100 fois et de 2711.3961058716154 pour une augmentation de 1000 fois. Ces ratios indiquent une efficacité beaucoup moins optimale en termes de temps d'exécution, avec une augmentation significative des temps lors des augmentations de charge.

Utilisation du CPU: 2/5
Le pourcentage d'utilisation du CPU est indiqué à 6% avant l'exécution et augmente à 10,5% pendant l'exécution. Bien que cette augmentation reste légère, elle suggère une charge de travail plus intense et une utilisation relativement plus élevée du CPU. Ainsi, l'impact global sur l'utilisation du CPU est assez important .

Consommation mémoire: 2/5
Le code utilise une liste pour stocker les mots extraits de la chaîne de caractères. Bien que la consommation mémoire dépende de la taille de la chaîne de caractères et du nombre de mots extraits, l'utilisation d'une liste peut entraîner une consommation de mémoire relativement plus élevée par rapport à d'autres structures de données plus optimisées, telles que les listes chaînées. Cela pourrait être une zone d'amélioration potentielle pour réduire la consommation de mémoire.