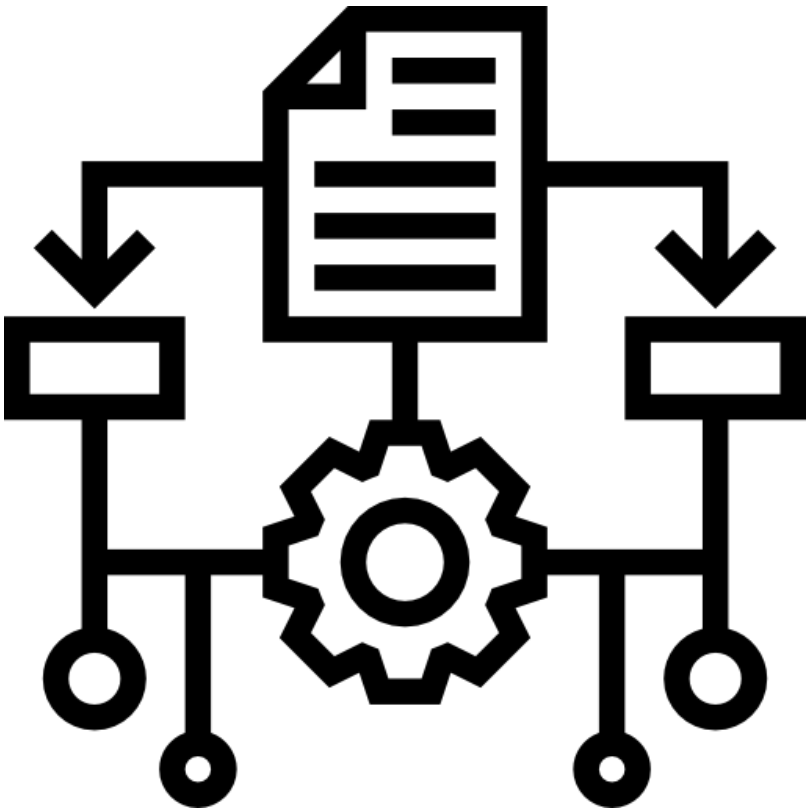
SAE 2.02 - Exploration algorithmique d'un problème

Rapport d’évaluation des algorithmes



Laguilliez Mathys 3B

Table des matières

1. Introduction :

Ce rapport d'évaluation des algorithmes permet de retrouver le travail de classification effectué pour les algorithmes RLE (Run-Length Encoding) qui m'ont été chargés d'évaluer. Il permet également de suivre le raisonnement qui m'a conduit à mes conclusions ainsi que les méthodes que j'ai employés pour la classification. Enfin, il précise comment reproduire les analyses effectuées sur les différentes catégories.

2. Méthodologie d'Évaluation

a) Simplicité

Afin de classer les différents algorithmes par simplicité, je voulais prendre en compte la lisibilité du code, ainsi que son bon fonctionnement. J'utilise les critères suivants :

- Nom des variables / fonctions clairs et significatif

- Présence de commentaires utiles

- Bonne utilisation de l'indentations

- Le code doit suivre les conventions de codage du langage utilisé

- Convention de nommages

- Réalisation des 4 méthodes à implémenter (2 RLE, 2 unRLE)

- Validité des tests JUnit[1]

- Une notation subjective en liens avec le Barème de notations des algorithmes « Barème pour noter les algorithmes.xlsx » fournies sur Webetud.

  (Fournie dans le répertoire avec le rapport)

[1] Lorsque le code est un autre langage que la java, les tests JUnit sont reproduits ou refait à la main si le code en permet la possibilité.

b) Efficacité

Afin de classer les différents algorithmes par efficacité d'exécution, je voulais prendre en compte uniquement des critères de respect de l'essentiel demander (implémentations des algorithmes de RLE/unRLE), et un critère de rapidité d'exécution. J'utilise les critères suivants :

- Réalisation des 4 méthodes à implémenter (2 RLE, 2 unRLE)

- La validité des tests JUnit fournies dans le projet GitHub ainsi que le celui que j'ai créés[2]

  (Disponible dans le répertoire analyse)

- La complexité algorithmique du programme

- Comparaisons des temps d'exécution[3] d'un tests JUnit testPerf () implémenter[4]

  (Disponible dans le répertoire analyse)

- Une notation subjective en liens avec le Barème de notations des algorithmes « Barème pour noter les algorithmes.xlsx » fournies sur Webetud.

  (Fournie dans le répertoire avec le rapport)

[2] Lorsque le code est un autre langage que la java, les tests JUnit sont reproduits ou refait à la main si le code en permet la possibilité.
[3] Moyenne de 5 temps afin d'avoir une valeur moins sensible à la variation du calcules de l'ordinateur utilisé
[4] Lorsque le code est un autre langage que la java, les tests JUnit sont reproduits ou refait à la main si le code en permet la possibilité.

c) Sobriété

Afin de classer les différents algorithmes par sobriété, il aurait fallu comparer l'utilisation en mémoires lors des différents test JUnit afin de pouvoir déterminer le quel algorithme consomme le moins de ressources lors de son exécution. Néanmoins n'arrivant pas à utiliser différentes méthodes afin de pouvoir réaliser cette analyse (Joular = cvc vide, VisualVM = jdk non détecter…) je me baserais donc uniquement sur la complexité algorithmique de chaque algorithme.

- Réalisation des 4 méthodes à implémenter (2 RLE, 2 unRLE)

- Validité des tests JUnit[5]

- La complexité algorithmique du programme

- Une notation subjective en liens avec le Barème de notations des algorithmes « Barème pour noter les algorithmes.xlsx » fournies sur Webetud.

  (Fournie dans le répertoire avec le rapport)

[5] Lorsque le code est un autre langage que la java, les tests JUnit sont reproduits ou refait à la main si le code en permet la possibilité.

3. Prérequis

Afin d'exécuter mes codes d'évaluations, voici ce que j'ai utilisé pour les réaliser :

- Java 21.0.1
- Python 3.12.3
- Beaucoup de patience

Dans le fichier pom.xml

- Compiler source de Maven : 19
- Compiler Target de Maven : 19
- JUnit version 4.13.2

Logiciel utilisé :

- Visual Studio Code

Extension utilisé pour les tests JUnit :

- Test Runner for Java

Dossiers utilisés pour importer les algorithmes à tester :

SAE 2.02\sae2024-2-02-laguilliez-mathys\src\main\java\iut\sae\algoTest

4. Reproductibilité

Pour reproduire mes analyses :

1. Implémenter l'algorithme à tester dans le dossier :

   « SAE 2.02\sae2024-2-02-laguilliez-mathys\src\main\java\iut\sae\algoTest »

2. Rajouter les deux tests JUnit présent dans le répertoire « Analyse » au fichier :

   « SAE 2.02\sae2024-2-02-laguilliez-mathys\src\test\java\iut\sae\algo\AlgoTest.java »

3. Importer les méthodes du fichier de l'algorithme dans le fichier JUnit avec l'utilisation de la commande :

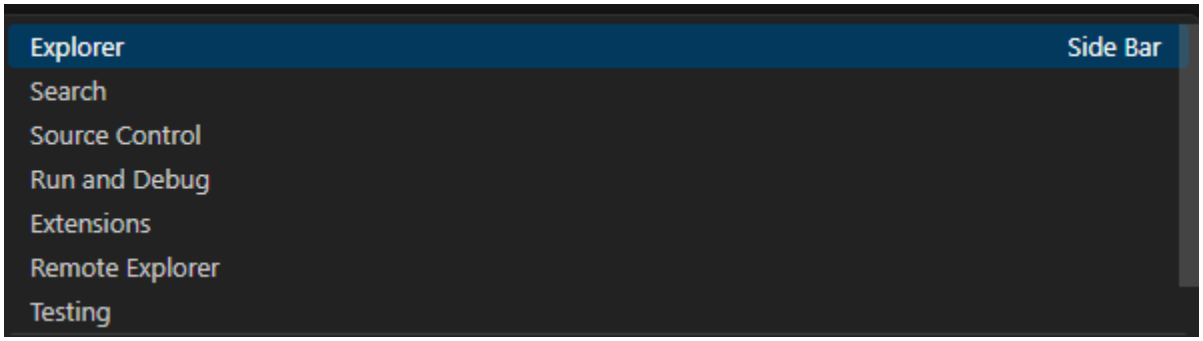
   « Import iut.sae. algoTest.insererNomClass; »

4. Modifier les AssertEquals (ctrl + f2 pour gagner du temps) sur le nom de la classe de la provenance de la méthode ainsi que le nom de la méthode

   Ex : assertEquals ("", *InsererNomClass.InsererNomMethode*("")) ;

5. Afin de récupérer le temps d'exécution du teste de performance testPerf (), il faut ouvrir l'onglet « Testing » disponible dans VS Code afin de récupérer le temps d'exécutions (accessible depuis

   Le logo  ou bien depuis le raccourcis ctrl + Q puis testing)



5. Evaluation des algorithmes

1) Simplicité :

NOM	Réalisation des 4 méthodes demander	Tests valides	Lisibilité du code	Respect des conventions de code	Note du code
46simplicite.java	NA	NA	NA	NA	Hors concours
63simplicite.java	4 / 4	Oui	2.5/3 Peu de commentaire	Oui	20
20simplicite.py	4 / 4	Non	3/3	Partiellement (Manque d'indentation)	NA (Boucle infinie ?)
50simplicite.java	4 / 4	Oui	2/3 Manque de commentaire	Oui	20
07simplicite.java	4 / 4	Oui	3 / 3	Oui	20

Classement

1er	07simplicite.java Le code est complet, passe tous les tests, très lisible et respecte les conventions java.
2ième	63simplicite.java Le code est complet, passe tous les tests avec un peu plus de temps.
3ième	50simplicite.java Le code est complet, passe tous les tests, commentaire trop léger, respecte les conventions java.
4ième	35efficacite.py Le code est complet, mais non fonctionnel et ne passe aucun test, Bien commenter, manque d'indentations.

2) Efficacité :

NOM	Réalisation des 4 méthodes demander	Tests valides	Complexité algorithmique	Test de Performance	Note du code
24efficacite.java	4/4	Oui	O(n)	14.9s	20
43efficacite.java	4/4	Non	O(n^2)	Echec	14
62efficacite.java	4/4	Oui	O(n)	0.99s	20
35efficacite.py	1/4	Non	O(n-1)	Non Evaluable	10
17efficacite.java	1/4	Non	O(n)	Non Evaluable	10

Classement

1er	62efficacite.java Le code est complet, passe tous les tests avec une forte rapidité d'exécution.
2ième	24efficacite.java Le code est complet, passe tous les tests avec un peu plus de temps.
3ième	43efficacite.java Le code est complet, petit défaut sur le unRLE itératif qui ne passe pas le test JUnit, ni le testPerf, échec du teste de performance, complexité algorithmique plus élever.
4ième	35efficacite.py Code partiel, seul la partie RLE (String in) est implémentée, pas de récursivité ni unRLE, passe 1 / 4 tests JUNIT fournis, Ne passe pas le testes de performance Possède une complexités algorithmique inférieur à 17efficacite.java
5ième	17efficacite.java Code partiel, seul la partie RLE (String in) est implémentée, pas de récursivité ni unRLE, passe 1 / 4 tests JUNIT fournis, Ne passe pas le testes de performance

3) Sobriété :

NOM	Réalisation des 4 méthodes demander	Tests valides	Complexité algorithmique	Note du code
50sobriete.java	4 / 4	Oui	O(s)	20
60sobriete.java	4 / 4	Oui	O(n)	20
44sobriete.c	NA	NA	NA	NA
54sobriete.java	2 / 4	2 / 4	O(n)	10

Classement

1er	**60sobriete.java** Le code est complet, passe tous les tests avec une plus forte rapidité d'exécution.
2ième	**50sobriete.java** Le code est complet, passe tous les tests avec un peu plus de temps.
3ième	**54sobriete.java** Le code est incomplet, manque unRLE.
4ième	**44sobriete.c** Non évalué, Incapacité a compilé du C sur ma machine.

6. Conclusion

Ce rapport d'évaluation des algorithmes m'a permis de comparer différents algorithmes de Run-Length Encoding (RLE) en fonction de plusieurs critères : l'efficacité, la simplicité et la sobriété. Cette expérience m'a fait prendre conscience de la complexité de la classification des algorithmes similaires, largement influencée par la méthodologie de test choisie pour évaluer le code.

Pour évaluer la simplicité des algorithmes, les critères de lisibilité, de structure conforme aux conventions de codage, et de commentaires clairs et pertinents ont été les plus important. Les algorithmes les mieux notés étaient ceux qui, tout en étant fonctionnels, étaient faciles à comprendre.

Efficacité et Sobriété

En ce qui concerne l'efficacité, les algorithmes les plus performants ont démontré une gestion optimale du temps d'exécution et une complexité algorithmique minimale, tout en réussissant les tests JUnit de manière satisfaisante.

Quant à la sobriété, bien que l'évaluation de la consommation de mémoire n'ait pas été possible en raison de limitations techniques, la complexité algorithmique a été utilisée comme indicateur principal.

En conclusion, Ce rapport offre une évaluation des performances et des caractéristiques des algorithmes de RLE, combinant des critères objectifs mesurables et une évaluation subjective de la lisibilité et de la maintenabilité du code. Il est à préciser que tout classement est influencé par les méthodes d'évaluation choisies et par les contraintes techniques rencontrées lors de l'évaluation.