

Compte-Rendu **Comparaison d'algorithmes**

R2
SAE2.02

Sommaire

Introduction	2
Notation des algorithmes	2
Explication des tests	3
Sobriété numérique	
Temps d'exécution	
Algorithme JAVA	4
Algorithme C	21
Algorithme Python	26

Introduction

Cette SAE se place dans un contexte pseudo-compétitif d'analyse d'algorithmes mutuel. La première semaine consiste en la rédaction d'un nombre compris entre un (1) et neuf (9) algorithmes rentrant dans l'un ou plusieurs des critères suivant au choix :

Efficacité, sobriété énergétique, et simplicité de compréhension, dans l'un des trois langages de programmation disponible :

C, JAVA, Python.

La seconde semaine consistera en l'analyse mutuelle des algorithmes rendus, selon les critères de :

- **Lisibilité du code**
 - Ce critère est subjectif. Il se base sur la facilité à comprendre ce que fait le code.
- **Qualité du code**
 - Vous utiliserez des outils open source de mesure de qualité de code (e.g., Codacy).
- **Efficacité**
 - Il s'agit d'évaluer la complexité algorithmique de la solution ($O(n^2)$ ou $O(n \log(n))$). Si on double par exemple la taille de la donnée en entrée, est-ce qu'on double le temps de calcul ?
- **Sobriété numérique**
 - Cela devient un critère de plus en plus important. Certains outils permettent de donner une mesure de la consommation en ressources d'un algorithme (e.g., Joular).
- **Temps d'exécution**
 - Il s'agit de mesurer le temps d'exécution.

Notation des algorithmes

Les algorithmes rendus seront notés de deux manières :

- La première sera une note objective basée sur si l'algorithme passe les tests selon le barème suivant :

Problème	Sanction	
Ne compile pas	Note finale /2	Exclusion du concours
Non respect de l'anonymat	-1	
Non respect de la consigne sur les méthodes de java.util (pour efficacité)	-1	Exclusion du concours
Non fonctionnel (retour erroné, ou pas du bon type attendu)	5/20	-
Fonctionnel mais ne passe pas les tests fournis initialement	10/20	-
Passe tous les tests fournis initialement	18/20	-
Passe vos tests supplémentaires plus complets	20/20	-

Une seconde note sera attribuée mais indicative à des fins de classements internes.

Explication des tests

Sobriété numérique :

Ce test détermine la consommation énergétique d'un programme, mesurée en micro joules (μJ).

Lorsque possible, ce test sera déterminé par les outils suivant :

- **Java**
 - *Pas de méthode découverte (La méthode $E(J) = P(W) * t(s)$ n'est pas fiable).*
- **Python**
 - pyRAPL, une bibliothèque python basé sur le processus RAPL (Running Average Power Limit) de Intel, déterminant la consommation énergétique entre l'appel de la fonction et la fin.
- **C**
 - *Pas de méthode découverte*

Temps d'exécution :

Ce test détermine le temps qui s'écoule entre l'appel de la fonction, et la réponse de la fonction.

Son unité est la microseconde (μs).

Lorsque possible, ce test sera déterminé par les outils suivant :

- **Java**
 - Méthode `RLERuntimeTest` écrite pour ce but. Ouvre un prompt en console demandant la méthode à tester (RLE/UNRLE).
Appelle 50 fois la fonction et mesure le temps que prend chaque réponse de fonction.
Le temps est ensuite divisé par 50 pour établir une moyenne.
Ci-dessous, le bloc de code significatif du test.

```
long start = System.nanoTime();
String result = "";
int intermediateTime = 0;

for (int i = 0; i < 50; i++){
    result = tester(in);

    intermediateTime += ((System.nanoTime() - start)/1000);
}

System.out.println("----- Résultats des tests -----");
System.out.println("Méthode testée : " + in);
if (in.equals("RLE")){
    System.out.println("Chaîne testée : " + STRINGRLE);
}else{
    System.out.println("Chaîne testée : " + STRINGUNRLE);
}
System.out.println("Résultat de l'opération : " + result);
System.out.println("Temps d'exécution pour 50 itérations : " + intermediateTime + "µs");
System.out.println("Temps d'exécution moyen : " + (intermediateTime/50) + "µs");
System.out.println("-----");
```

- **Python**
 - librairie `time`.
Récupère le temps au début de l'appel avec `time.time()`, puis similairement à Java, appelle 50 fois la fonction et ajoute le temps que prend chaque appel.
- **C**
 - Librarie `time.h`.
Bien que rudimentaire, renvoie similairement à Python et java le temps qui s'est écoulé entre l'appel et la réponse de la méthode.
Sa seule contrainte est qu'elle est limitée en microsecondes et ne peut pas mesurer jusqu'au nano.

Algorithmes JAVA

57simplicite		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Non complétés : UnRLE, UnRLERecuratif Erreur : junit.framework.ComparisonFailure: expected: [[ab]c] but was:[[1a1b1]c] at iut.sae.algo.AlgoTest.testUnRLE(AlgoTest.java:69)	9/20
Notation de la qualité*		
Lisibilité du code	L'abondance de commentaires entravent paradoxalement la lisibilité du texte de part leur présence et leur placement	2.5/5
Qualité du code	Très bonne qualité du code, seule deux (2) améliorations mineures peuvent être effectuées. (Voir annexe 1)	5/5
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : $O(n)$ unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $\sim O(\sum_0^{n/2} i)$	4.75/5
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : $\sim 2.859ms$ unRLE : Pour une chaîne de longueur 10 une fois décompressée : $\sim 3.222ms$	3/5

Annexe

Annexe 1 : Commentaires détaillés sur la qualité du code.

① Current 2 Ignored 0

Filter by Language Severity Category 1 Pattern Author Clear all

MINOR Code style

The static method name 'RLE' doesn't match '[a-z][a-zA-Z0-9]*'

src/main/java/iut/sae/algo/Algo.java

```
5 public static String RLE(String in){
```

MINOR Code style

The static method name 'RLE' doesn't match '[a-z][a-zA-Z0-9]*'

src/main/java/iut/sae/algo/Algo.java

```
18 public static String RLE(String in, int iteration) throws AlgoException{
```

Annexe 2 : Résultats des tests

----- Résultats des tests -----	----- Résultats des tests -----
Méthode testée : RLE	Méthode testée : UNRLE
Chaine testée : aaabbzzzed	Chaine testée : 3a2b3z1e1d
Résultat de l'opération : 3a2b3z1e1d	Résultat de l'opération : 333a22b333z1e1d
Temps d'exécution pour 50 itérations : 142955µs	Temps d'exécution pour 50 itérations : 161100µs
Temps d'exécution moyen : 2859µs	Temps d'exécution moyen : 3222µs

Retours généraux

Le code ne passe pas les exigences minimum demandées. La méthode `unRLE` est incapable de décoder une chaîne fournie en argument, ce qui inclut que `unRLERecuratif` possède le même problème. Le code fournie ne pose cependant pas de problèmes notables en terme de qualité. En tant qu'algorithme de simplicité, certaines méthodes sont employés qui aurait pu être évitées afin de simplifier la compréhension générale du code et éviter les détours (Matcher, pattern). Le temps d'exécution est correct pour de la simplicité.

Note finale

Algorithme	Qualité	Rang
09/20	15.25/20	5ème

59simplicite		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Non complétés : RLE Erreur : junit.framework.ComparisonFailure: expected: [9W[4]W] but was:[9W[5]W]	10/20
Notation de la qualité*		
Lisibilité du code	Excellente lisibilité, commentaires clairs et bien placés	5/5
Qualité du code	Bonne qualité de code. Une petite erreur d'optimisation (Voir annexe 1)	5/5
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : $O(n)$ unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $\sim O(\sum_0^{n/2} i)$	4.75/5
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : $\sim 3.179ms$ unRLE : Pour une chaîne de longueur 10 une fois décompressée : $\sim 1.274ms$	3.5/5

Annexe

Annexe 1 : Commentaires détaillés sur la qualité du code.



Annexe 2 : Résultats des tests

----- Résultats des tests -----	----- Résultats des tests -----
Méthode testée : RLE	Méthode testée : UNRLE
Chaine testée : aaabbzzzed	Chaine testée : 3a2b3z1e1d
Résultat de l'opération : 3a2b3z1e1d	Résultat de l'opération : aaabbzzzed
Temps d'exécution pour 50 itérations : 158996µs	Temps d'exécution pour 50 itérations : 63747µs
Temps d'exécution moyen : 3179µs	Temps d'exécution moyen : 1274µs

Retours généraux

Le code ne passe pas les exigences minimum demandées. La méthode `RLE` possède un problème d'encodage des chaînes, en l'occurrence, il rajoute un caractère qui n'est pas censé exister. Le code fourni est cependant très lisible avec des commentaires placés juste comme il faut pour comprendre sans obstruer la visibilité. Une petite erreur d'optimisation négligeable mais à ne pas multiplier. En tant qu'algorithme de simplicité, le code est épuré et va droit au but, très facile à comprendre sans multiplier les classes ou objets.

Note finale

Algorithme

10/20

Qualité

18.25/20

Rang

4ème

28simplicite		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Non complétés : unRLE, unRLERecuratif manquant	8/20
Notation de la qualité*		
Lisibilité du code	Bonne lisibilité	4/5
Qualité du code	Très bonne qualité de code.	5/5
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : 0(n) unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : MISSING	2.5/5
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : ~1.361ms unRLE : Pour une chaîne de longueur 10 une fois décompressée : MISSING	2.5/5

Annexe

Annexe 1 : Résultat des tests.

```
----- Résultats des tests -----  
Méthode testée : RLE  
Chaine testée : aaabbzzzed  
Résultat de l'opération : 3a2b3z1e1d  
Temps d'exécution pour 50 itérations : 68070µs  
Temps d'exécution moyen : 1361µs
```

Retours généraux

Le code ne passe pas les exigences minimum demandées. La méthode `unRLE` est manquant, ce qui inclut que `unRLERecuratif` ne peut pas exister. Le code fournie ne pose cependant pas de problèmes notables en terme de qualité et est lisible. En tant qu'algorithme de simplicité, les méthodes utilisées sont simples à comprendre et entretenir, tout en étant rapide.

Note finale

Algorithme

08/20

Qualité

14/20

Rang

7ème

49efficacite		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Correct	20/20
Notation de la qualité*		
Lisibilité du code	Bon code, un peu compact.	4/5
Qualité du code	Très bon code. Pas d'erreurs notables à souligner.	5/5
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : $O(n)$ unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $\sim O(\sum_0^n i)$	4.5/5
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : $\sim 13.720ms$ unRLE : Pour une chaîne de longueur 10 une fois décompressée : $\sim 10.314ms$	1/5

Annexe

Annexe 1 : Résultats des tests

----- Résultats des tests -----	----- Résultats des tests -----
Méthode testée : RLE	Méthode testée : UNRLE
Chaine testée : 3a2b3z1e1d	Chaine testée : 3a2b3z1e1d
Résultat de l'opération : 3a2b3z1e1d	Résultat de l'opération : aaabbzzzed
Temps d'exécution pour 50 itérations : 686036µs	Temps d'exécution pour 50 itérations : 515703µs
Temps d'exécution moyen : 13720µs	Temps d'exécution moyen : 10314µs

Retours généraux

Le code respecte les tests demandés et ne pose pas de problème en terme de qualité. Le code est compact ce qui le rend un peu plus dur à lire mais l'objectif est l'efficacité plus que la simplicité de lecture. Le code ne possède pas de problèmes majeurs à souligner. Cependant le nombre d'itération peut être divisé par 2 puisque nous savons que dans un encodage RLE, l'information des chiffres est toujours stockée dans des rangs pairs (0-2-4-6...), ce qui permet une augmentation de i de 2 plutôt que 1.

De plus, le temps de réponse comparé aux autres algorithmes, même dans la catégorie simplicité ou sobriété, est notablement plus élevée ce qui pose problème dans un algorithme soumis dans la catégorie efficacité.

Note finale

Algorithme	Qualité	Rang
20/20	14.5/20	6ème

16efficacite		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Correct	20/20
Notation de la qualité*		
Lisibilité du code	Excellente lisibilité, commentaires clairs et bien placés	5/5
Qualité du code	Très bon code. Pas d'erreurs notables à souligner.	5/5
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : $O(n)$ unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $\sim O(n/2)$	5/5
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : $\sim 0.897ms$ unRLE : Pour une chaîne de longueur 10 une fois décompressée : $\sim 1.124ms$	5/5

Annexe

Annexe 1 : Résultats des tests

----- Résultats des tests -----	----- Résultats des tests -----
Méthode testée : RLE	Méthode testée : UNRLE
Chaine testée : aaabbzzzed	Chaine testée : 3a2b3z1e1d
Résultat de l'opération : 3a2b3z1e1d	Résultat de l'opération : aaabbzzzed
Temps d'exécution pour 50 itérations : 44899µs	Temps d'exécution pour 50 itérations : 56219µs
Temps d'exécution moyen : 897µs	Temps d'exécution moyen : 1124µs

Retours généraux

Excellent code. Les tests sont tous corrects, le code est très lisible avec des commentaires clairs et bien placés. Le code ne possède pas de problèmes majeurs de qualité, avec une complexité algorithmique optimale de $n/2$ pour décoder les chaînes, ce qui se répercute sur le temps d'exécution du script très bas.

Note finale

Algorithme

20/20

Qualité

20/20

Rang

1er

30efficacite		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Correct	20/20
Notation de la qualité*		
Lisibilité du code	Excellente lisibilité, commentaires clairs et bien placés	5/5
Qualité du code	Très bon code. Pas d'erreurs notables à souligner.	5/5
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : $O(n)$ unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $\sim O(\sum_0^{n/2} i)$	4.75/5
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : $\sim 1.033ms$ unRLE : Pour une chaîne de longueur 10 une fois décompressée : $\sim 1.263ms$	5/5

Annexe

Annexe 1 : Résultat des tests

```
----- Résultats des tests -----  
Méthode testée : RLE  
Chaine testée : aaabbzzzed  
Résultat de l'opération : 3a2b3z1e1d  
Temps d'exécution pour 50 itérations : 51679µs  
Temps d'exécution moyen : 1033µs
```

```
----- Résultats des tests -----  
Méthode testée : UNRLE  
Chaine testée : 3a2b3z1e1d  
Résultat de l'opération : aaabbzzzed  
Temps d'exécution pour 50 itérations : 63174µs  
Temps d'exécution moyen : 1263µs
```

Retours généraux

Très bon code. Les tests sont tous positifs, et la classe fournie est très lisible avec des commentaires judicieusement placés. Pas d'erreurs notables à souligner. Son nombre d'itération est correct pour un algorithme efficacité, ce qui se répercute sur son temps d'exécution faisant partie des plus bas.

Note finale

Algorithme

20/20

Qualité

19.75/20

Rang

2ème

55efficacite		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Correct	20/20
Notation de la qualité*		
Lisibilité du code	Bonne lisibilité	4.5/5
Qualité du code	Très bon code. Pas d'erreurs notables à souligner.	5/5
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : $O(n)$ unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $\sim O(\sum_0^n i)$	4.5/5
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : $\sim 1.504ms$ unRLE : Pour une chaîne de longueur 10 une fois décompressée : $\sim 1.667ms$	4/5

Annexe

Annexe 1 : Résultats des tests

```
----- Résultats des tests -----  
Méthode testée : RLE  
Chaine testée : aaabbzzzed  
Résultat de l'opération : 3a2b3z1e1d  
Temps d'exécution pour 50 itérations : 75239µs  
Temps d'exécution moyen : 1504µs
```

```
----- Résultats des tests -----  
Méthode testée : UNRLE  
Chaine testée : 3a2b3z1e1d  
Résultat de l'opération : aaabbzzzed  
Temps d'exécution pour 50 itérations : 83377µs  
Temps d'exécution moyen : 1667µs
```

Retours généraux

Le code respecte les tests demandés et ne pose pas de problème en terme de qualité. La lisibilité est bonne ce qui permet une compréhension efficace du programme. Le code ne possède pas d'erreurs notables. Sa complexité algorithmique est très satisfaisante, sauf pour la méthode `unRLE` où une itération `i += 2` serait plus judicieuse qu'une en `i++` simple. Le temps d'exécution pour un programme soumis dans la catégorie efficacité est bien.

Note finale

Algorithmme

20/20

Qualité

18/20

Rang

3ème

15sobriete		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Ce n'est pas à moi d'écrire la fonction RLE et unRLE récursif... Aucun des tests ne passe.	5/20
Notation de la qualité*		
Lisibilité du code	Bonne lisibilité	4.5/5
Qualité du code	Bon code, pour ce qui est disponible. Veiller à bien se débarrasser des variables inutiles (voir annexe 1)	4.5/5
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : MISSING unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $\sim O(\sum_0^n i)$	2/5
Sobriété numérique		-/20
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : MISSING unRLE : Pour une chaîne de longueur 10 une fois décompressée : NotYetImplemented	0.25/5

Annexe

Annexe 1 : Commentaires détaillés sur le code



Retours généraux

Ce « programme » est une blague. Aucune des fonctionnalités demandées n'est là, et même si elles le sont, elles ne font même pas correctement ce qui en est demandé. Par conséquent, il est impossible de juger la complexité algorithmique, le temps d'exécution...
Fort soupçon de copier-coller de ChatGPT. Plus d'effort aurait pu être investi pour le temps qui a été accordé.

Note finale

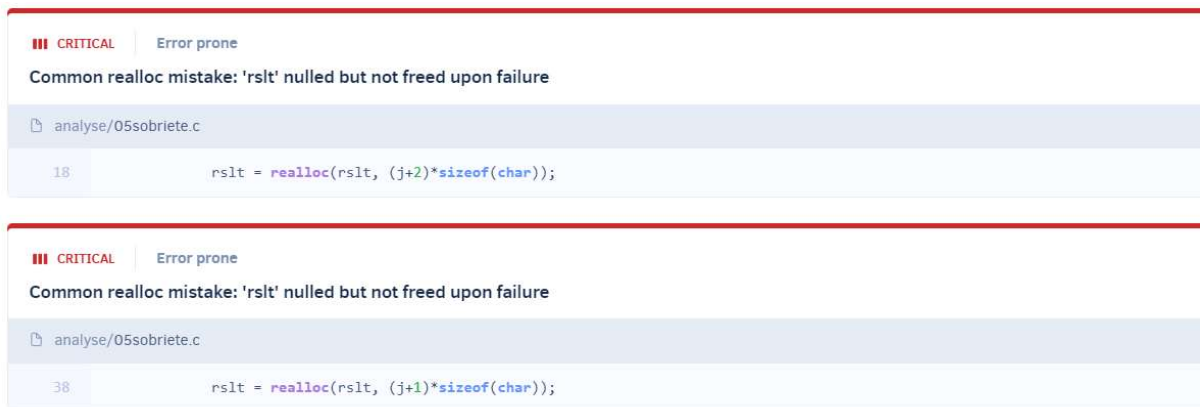
Algorithmme	Qualité	Rang
05/20	11.25/20	8/8

Algorithmes C

05sobriete.c		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Correct.	20/20
Notation de la qualité*		
Lisibilité du code	Bonne lisibilité pour du C	4.5/5
Qualité du code	Fuites mémoires à régler. Bien penser à désallouer les emplacements mémoires. (voir annexe)	2.5/5
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : $O(n)$ unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $\sim O(\sum_0^{n/2} i)$	5/5
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : $<0\mu s$ unRLE : Pour une chaîne de longueur 10 une fois décompressée : $<0\mu s$	5/5

Annexe

Annexe 1 : Commentaires détaillés sur le code.



Retours généraux

Très bon code. Passe les tests demandés et est lisible pour du C. La complexité algorithmiques pour un langage bas niveau est très satisfaisante et ne pourrait être très difficilement abaissée. Le temps d'exécution est assez bas pour ne pas mesurable car elle la précision en nanosecondes n'est facilement mesurable en C. Cependant pour de la sobriété veillez à bien corriger les fuites mémoires qui consomme de la mémoire, et donc de l'énergie. Autre ça, très bons algorithmes.

Note finale

Algorithme

20/20

Qualité

17/20

Rang

2ème

39sobriete.c		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Correct.	20/20
Notation de la qualité*		
Lisibilité du code	Bonne lisibilité pour du C	4.5/5
Qualité du code	Bon code. Quelques erreurs de logique d'optimisation à régler mais le but de l'algorithme n'est pas l'efficacité. (voir annexe 1)	3.5/5
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : $O(n)$ unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $\sim \sum_0^{n/2} i$	5/5
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : $<0\mu s$ unRLE : Pour une chaîne de longueur 10 une fois décompressée : $<0\mu s$	5/5

Annexe

CRITICAL

Security Input Validation

Use sprintf_s, snprintf, or vsnprintf instead.

analyse/39sobriete.c

21

if (in[i] == char_precedent) {

22

if (nombre_repetitions == 9) {

23

sortie_intermediaire += sprintf(sortie_intermediaire, "%d%c", nombre_repetitions, char_precedent);

24

nombre_repetitions = 1;

25

} else {

MEDIUM

Error prone

Either the condition 'in==NULL' is redundant or there is possible null pointer dereference: in.

analyse/39sobriete.c

42

return strdup(in);

Retours généraux

Très bon code. Les algorithmes sont clairs pour du C, mais possèdent quelques problèmes d'optimisation pour un algorithme en sobriété. La complexité algorithmique est optimale pour du C, ce qui se répercute également sur son temps d'exécution.

Note finale

Algorithme

20/20

Qualité

18/20

Rang

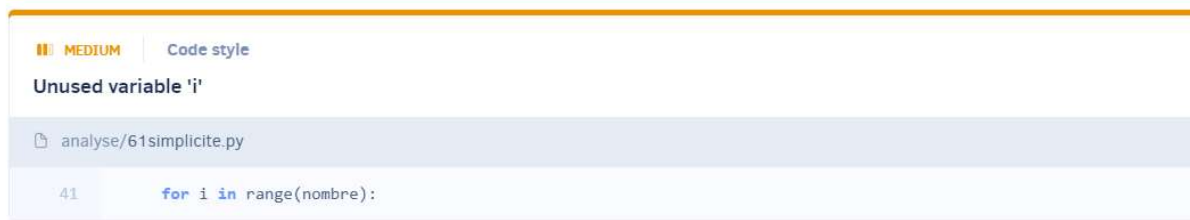
1^{ère}

Algorithmes Python

61simplicite.py		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Correct.	20/20
Notation de la qualité*		
Lisibilité du code	Très lisible	4/4
Qualité du code	Bonne qualité. Faire attention aux variables inutiles. (Voir annexe 1)	3.5/4
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : $O(n)$ unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $O(n/2)$	4/4
Sobriété numérique	RLE : Consommation énergétique de l'algorithme pour renvoyer un résultat : PKG : socket 0 : 4821.0000 μJ DRAM : socket 0 : 855.0000 μJ RLE : Consommation énergétique de l'algorithme pour renvoyer un résultat : PKG : socket 0 : 5982.0000 μJ DRAM : socket 0 : 488.0000 μJ	3.75/4
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : $\sim 0.707ms$ unRLE : Pour une chaîne de longueur 10 une fois décompressée : $\sim 0.633ms$	3/4

Annexe

Annexe 1 : Commentaires détaillés sur le code



Retours généraux

Très bon code. Le choix du python pour un algorithme en simplicité de compréhension est judicieux, ce qui fait que son contenu est très lisible et facilement compréhensible. Bonne qualité de code, avec juste un petit problème de variable inutilisée (voir annexe 1). Le temps d'exécution pour du python est très correct.

Note finale

Algorithme

20/20

Qualité

18.25/20

Rang

3ème

27simplicite.py		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Correct.	20/20
Notation de la qualité*		
Lisibilité du code	Très compréhensible	4/4
Qualité du code	Pas de problèmes notables. Bonne qualité.	4/4
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : $O(n)$ unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $O(n/2)$	4/4
Sobriété numérique	RLE : Consommation énergétique de l'algorithme pour renvoyer un résultat : PKG : socket 0 : 4394.0000 μJ DRAM : socket 0 : 549.0000 μJ unRLE : PKG : socket 0 : 7690.0000 μJ DRAM : socket 0 : 671.0000 μJ	3/4
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : $\sim 0.230ms$ unRLE : Pour une chaîne de longueur 10 une fois décompressée : $\sim 0.839ms$	3.5/4

Annexe

Retours généraux

Code très compréhensible, la qualité du code ne pose pas non plus de problème, et sa complexité algorithmique est très satisfaisante. Très bon temps d'exécution. Pour un algorithme soumis en simplicité, le code remplit parfaitement les critères de lisibilité.
Code très satisfaisant.

Note finale

Algorithme

20/20

Qualité

18.5/20

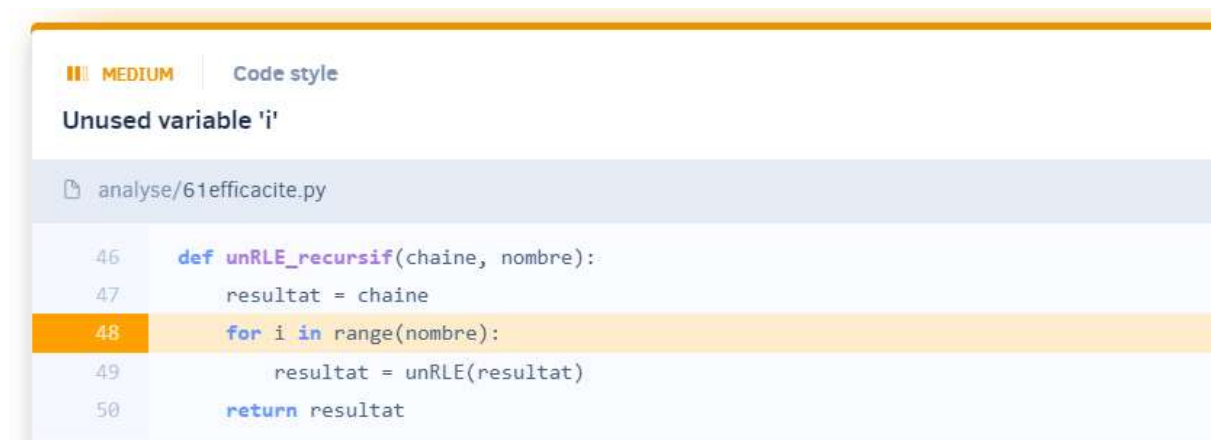
Rang

2ème

61efficacite.py		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	Correct.	20/20
Notation de la qualité*		
Lisibilité du code	Très compréhensible	4/4
Qualité du code	Bonne qualité. Faire attention aux variables inutiles. (Voir annexe 1)	4/4
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : $O(n)$ unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : $O(n/2)$	4/4
Sobriété numérique	RLE : Consommation énergétique de l'algorithme pour renvoyer un résultat : PKG : socket 0 : 4394.0000 μJ DRAM : socket 0 : 549.0000 μJ unRLE : PKG : socket 0 : 7630.0000 μJ DRAM : socket 0 : 1160.0000 μJ	3/4
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : $\sim 0.689ms$ unRLE : Pour une chaîne de longueur 10 une fois décompressée : $\sim 1.118ms$	3/4

Annexe

Annexe 1 : Commentaires détaillés sur le code.



```
46 def unRLE_recuratif(chaine, nombre):
47     resultat = chaine
48     for i in range(nombre):
49         resultat = unRLE(resultat)
50     return resultat
```

The screenshot shows a code editor with a yellow header bar. Below the header, there's a lint error notification: 'MEDIUM Code style Unused variable \'i\'' with a yellow background. Below this, the file path 'analyse/61efficacite.py' is shown. The code itself is in a light blue editor with line numbers 46 to 50. Line 48, which contains the line 'for i in range(nombre):', is highlighted in yellow to indicate the error.

Retours généraux

Très bon code. Passe les tests d'encodage et de décodage sans problèmes particuliers. Le code est très compréhensible, et de bonne qualité. La variable inutilisée ne pose pas réellement de problème puisque itérer sans variable pose un problème. La complexité algorithmique est optimale dans cette catégorie. Cependant, pour un algorithme soumis en efficacité, le temps d'exécution de la méthode de décodage peut être retravaillé, puisqu'il est plus élevé que des algorithmes sobres du même langage. De même, python étant un langage interprété, le temps d'exécution en sera impacté.

Note finale

Algorithme	Qualité	Rang
20/20	19/20	1er

35sobriete.py		
Notation des algorithmes		
Test	Résultat	Note
Test JUnit	RLE NON CONFORME, unRLE MANQUANT « aaaaaaaaaa » donne 10a au lieu de 9a1a	8/20
Notation de la qualité*		
Lisibilité du code	Peu compréhensible	2/4
Qualité du code	Pas de problèmes notables.	4/4
Efficacité algorithmique	RLE : Nombre d'itération de l'algorithme pour renvoyer un résultat : 0(n) unRLE : Nombre d'itération de l'algorithme pour décompresser un string (Soit i un chiffre composant la séquence $N \in (1,9)$) : MISSING	2/4
Sobriété numérique	RLE : Consommation énergétique de l'algorithme pour renvoyer un résultat : PKG : socket 0 : 5066.0000 μJ DRAM : socket 0 : 488.0000μJ unRLE : MISSING	1.5/4
Temps d'exécution	RLE : Pour une chaîne de longueur 10 non compressée : ~0.813ms unRLE : Pour une chaîne de longueur 10 une fois décompressée : MISSING	2/4

Annexe

Retours généraux

Code non conforme. La méthode `RLE` ne s'arrête pas comme demandée au bout du 10ème caractère afin de les grouper de la manière « 9a6a », ajoutera le nombre de fois que le caractère est présent sans limite (« 15a » par exemple). La méthode `unRLE` est quant à elle absente. Le code est peu lisible et semble bâclé. Le peu de contenu ne mène pas à des problèmes de code cependant. La complexité algorithmique pour l'encodage est satisfaisant, tout comme son temps d'exécution.

Note finale

Algorithmme

08/20

Qualité

11.5/20

Rang

4ème