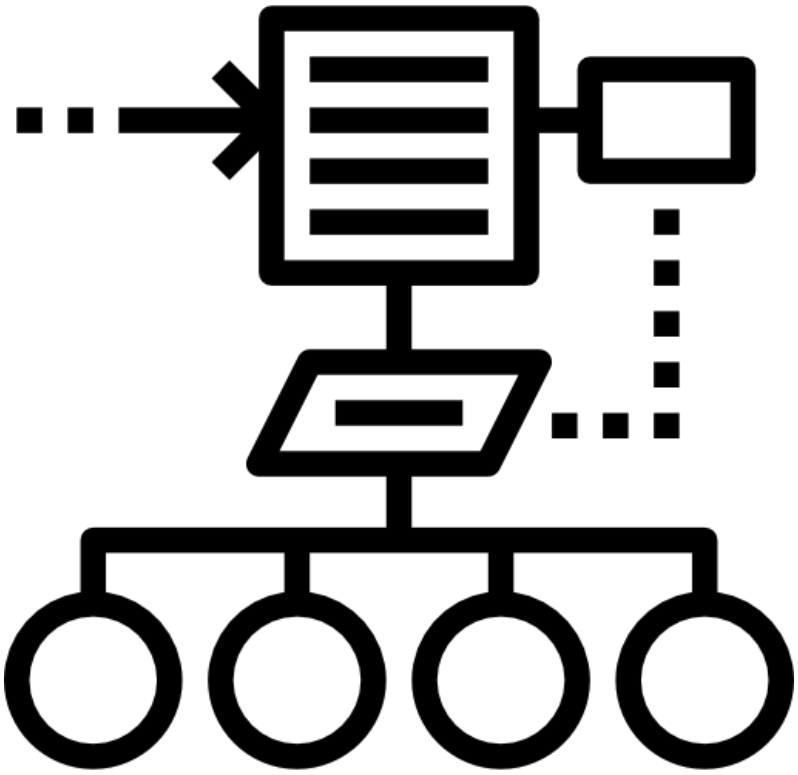
WINTERHOFF Ophélie 3B

SAE 2.02 : Exploration Algorithmique d'un problème

Partie 2



INTRODUCTION :

Ce document représentant la partie 2 de la SAE 2.02 – Exploration Algorithmique d'un problème, présente une analyse comparative d'algorithmes par catégorie (Efficacité, Simplicité et Sobriété). Nous avons évalué leur performance, leur complexité, leur efficacité, leur lisibilité et autres.

Nous commencerons par voir les Informations supplémentaires avant de passer aux comparaisons des algorithmes.

I. Informations Complémentaires :

1. Prérequis

Avant d'exécuter mes codes d'évaluation, assurez-vous d'avoir les éléments suivants :

- Un Environnement de développement intégré (IDE) comme par exemple :
  - Visual Studio Code (VS Code)
  - IntelliJ IDEA
  - Replit (solution d'IDE sur navigateur -> utile pour le C)
- Java : Version 21 ou 17 installée
- Python : Version 3.11.9 par exemple

- Pour exécuter le C (ainsi que les tests), télécharger les fichiers de chaque algo présent dans le dossier analyse et les insérés dans un Rep initialisé sur Replit.
- Pour exécuter le python (ainsi que les tests), télécharger les fichiers de tests et le main (à copier directement dans le fichier où les algorithmes sont présents) présent dans le dossier analyse et utiliser VS Code pour l'exécution (ou un autre IDE).
- Pour exécuter le Java, télécharger le main et les tests présents dans le dossier analyse dans le même dossier que l'algorithme. Puis importer l'algorithme dans chaque fichier.

2. Évaluation des algorithmes

Les algorithmes sont notés en 2 parties. La première partie est basée sur le fonctionnement du code dans sa globalité et le passage des tests associés. Et la deuxième partie est basée sur l'évaluation des critères de comparaisons des algorithmes.

Première partie :

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

La deuxième partir porte sur l'évaluation de quelques critères :

- La lisibilité
- La qualité
- L'efficacité
- Le temps d'exécution

3. Tests et outils pour les critères de comparaison

- **Qualité :**
  - Outil : utilisation de Codacy qui est un outil d'analyse statique du code, de l'algorithme
- **Temps d'exécution :**
  - ➢ Java :
  - Récupération du temps d'exécution de RLE et/ou unRLE dans le main de RLEConsole.java. On utilise la méthode **System.nanoTime()**, pour capture les horodatages avant et après l'exécution d'un algorithme de manière très précise. Puis on calcule le temps d'exécution en soustrayant l'heure capturée en nanosecondes après l'exécution et l'heure avant l'exécution.

```java
System.out.println("Entrée : " + in);

long startTime = System.nanoTime();
String output = Algo27sobriete.unRLE(in);
long endTime = System.nanoTime();

System.out.println("Sortie : " + output);

long dureeNano = endTime - startTime;

System.out.println("Temps d'exécution : " + dureeNano + " ns (" + (dureeNano / 1_000_000.0) + " ms)");
```

  - Récupération du temps d'exécution d'une décompression avec itération, d'une compression avec le même nombre d'itérations, grâce à un test JUnit (quand on l'exécute on a le temps à gauche dans VSCode)

```java
@Test
public void testPerformance() {
    try{
        assertEquals(expected:"wwwbzzzzzzzzz", Algo62sobriete.unRLE(Algo62sobriete.RLE(in:"wwwbzzzzzzzzz",iteration:30), iteration:30));
    }catch(Exception e){
        fail(message:"Exception inatendue");
    }
}
```

  - ➢ Python
  - Pour la récupération du temps d'exécution des algorithmes RLE et unRLE, j'ai réalisé la même chose que pour Java dans le main du fichier, mais en utilisant la méthode **time.perf_counter()**

```python
print("Entrée : " + in_string)

start_time = time.perf_counter()
output = simplicite.unRLE(in_string)
end_time = time.perf_counter()

print("Sortie : " + output)

duration_seconds = end_time - start_time
duration_millis = duration_seconds * 1000.0

print(f"Temps d'exécution : {duration_seconds:.6f} s ({duration_millis:.3f} ms)")
```

  - ➢ C
  - Pour la récupération du temps d'exécution des algorithmes RLE et unRLE, j'ai réalisé la même chose que pour Java dans le main du fichier, mais en utilisant la fonction **clock_gettime()** avec l'horloge **CLOCK_MONOTONIC**

```c
printf("Entree : %s\n", in);

struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);
char *output = RLE(in);
clock_gettime(CLOCK_MONOTONIC, &end);

printf("Sortie : %s\n", output);

double durationMillis = (end.tv_sec - start.tv_sec) * 1000.0 + (end.tv_nsec - start.tv_nsec) / 1000000.0;

printf("Temps d'execution : %.3f ms\n", durationMillis);
```

- Sobriété
  - Concernant la sobriété énergétique, il m'a été impossible de réaliser une comparaison de consommation énergétique. La consommation d'énergie d'un algorithme dépend de nombreux facteurs tels que les composants matériels, leur puissance, l'environnement de développement (IDE), la mémoire disponible, etc. Les résultats varieraient donc en fonction de chaque machine utilisée. De plus, la formule E=P*t (énergie = puissance * temps) n'est pas applicable ici car elle nécessite la mesure précise de la puissance des composants pendant l'exécution, ce qui est difficile à obtenir et peut être très variable.

II. Évaluation et Classement des Algorithmes par catégorie :

1. Efficacité

1.1. 21efficacite.java

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat et respect des consignes quant aux méthode de java.util Compile et chaque algorithme fonctionne	10/10
Tests JUnit + Test supplémentaire	Passe tous les tests et le test supplémentaire	10/10
Critères de comparaison		
Lisibilité	Bonne lisibilité : noms de variables explicites	5/5
Qualité	Pas de problème à signaler ; Très bonne qualité	5/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère de la chaîne d'entrée une seule fois unRLE : O(n*m) Il parcourt chaque paire de caractères et ajoute les occurrences	4/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 0,6069ms unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 0,7646ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça met 38ms	5/5
TOTAL Notation avec le barème donné		20/20
TOTAL Critère de comparaison		19/20
MOYENNE		19,5/20

1.2. 39efficacite.java

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat et respect des consignes quant aux méthode de java.util Compile et chaque algorithme fonctionne	10/10
Tests JUnit + Test supplémentaire	Passe tous les tests et le test supplémentaire	10/10
Critères de comparaison		
Lisibilité	Bonne lisibilité : noms de variables explicites	5/5
Qualité	Pas de problème à signaler ; Très bonne qualité	5/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère unRLE : O(n) parcourt chaque paire de caractères	4,5/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 0,7399ms unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 0,8312ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça met 42ms	5/5
TOTAL Notation avec le barème donné		20/20
TOTAL Critère de comparaison		19,5/20
MOYENNE		19,75/20

1.3. 42efficacite.java

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat et respect des consignes quant aux méthode de java.util Compile et chaque algorithme fonctionne	10/10
Tests JUnit + Test supplémentaire	Passe tous les tests et le test supplémentaire	10/10
Critères de comparaison		
Lisibilité	Assez bonne lisibilité : noms de variables à revoir comme parce exemple c, et manque d'aération	3,5/5
Qualité	Bonne qualité. Petites erreurs quant aux réaffectations des paramètres à l'intérieur des méthodes qui seraient à éviter	3,8/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère unRLE : O(n) parcourt chaque paire de caractères	4,5/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 0,7877ms unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 0,6635ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça met 37ms	5/5
TOTAL Notation avec le barème donné		20/20
TOTAL Critère de comparaison		17,8/20
MOYENNE		18,9/20

1.4. 63efficacite.java

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat et respect des consignes quant aux méthode de java.util Compile et chaque algorithme fonctionne	10/10
Tests JUnit + Test supplémentaire	Passe tous les tests et le test supplémentaire	10/10
Critères de comparaison		
Lisibilité	Assez bonne lisibilité : noms de variables explicites et présence de quelques commentaires, mais manque d'aération	4,8/5
Qualité	Bonne qualité	4,8/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère unRLE : O(n) parcourt chaque paire de caractères	4,5/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 0,9643ms unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 0,7463ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça met 39ms	5/5
TOTAL Notation avec le barème donné		20/20
TOTAL Critère de comparaison		19,1/20
MOYENNE		19,55/20

1.5. 68efficacite.java

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat et respect des consignes quant aux méthode de java.util Compile et chaque algorithme fonctionne	10/10
Tests JUnit + Test supplémentaire	Passe tous les tests et le test supplémentaire	10/10
Critères de comparaison		
Lisibilité	Assez bonne lisibilité : présence de commentaires (voire un peu trop), et certains noms de variables à revoir comme par exemple c	3,8/5
Qualité	Pas de problème à signaler ; Très bonne qualité	5/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère unRLE : O(n) parcourt chaque paire de caractères	4,5/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 0,8209ms unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 0,806ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça met 38ms	5/5
TOTAL Notation avec le barème donné		20/20
TOTAL Critère de comparaison		18,3/20
MOYENNE		19,15/20

1.6. Classement

CLASSEMENT EFFICACITÉ	
Podium	Nom de l'algorithme
1er	**39efficacite.java** : Excellent code et les tests sont tous passés. La qualité, la lisibilité et l'efficacité sont excellents et temps d'exécution très rapide
2ème	**63efficacite.java** : Le code est très bon et les tests sont tous passés. Cependant la qualité et la lisibilité sont très légèrement inférieurs aux 2 premiers. Mais l'efficacité et le temps d'exécution sont aussi très bons
3ème	**21efficacite.java** : Excellent code et les tests sont tous passés. La qualité et la lisibilité sont excellents, sans oublier le temps d'exécution qui est très rapide. Cependant pour la unRLE l'efficacité n'est pas la meilleure qui soit
4ème	**68efficacite.java** : Tout est très bon, les tests aussi sont tous passés. La qualité, l'efficacité et le temps d'exécution sont aussi très, très bons. La seule chose qui est moins bien que les autres c'est la lisibilité (mauvais noms de variables et trop de commentaires
5ème	**42efficacite.java** : C'est aussi très bon, et tous les tests sont aussi passés. Mais les problèmes (si on peut appeler ça des problèmes) c'est des petites erreurs quant à la qualité du code (réaffectation) et quant à la lisibilité (manque d'aération et noms de variables peu explicites). Cependant le temps d'exécution et l'efficacité sont aussi bons que les autres (temps très légèrement inférieur au premier)

2. Simplicité

2.1. 11simplicite.java

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat Compile et chaque algorithme fonctionne	10/10
Tests JUnit + Test supplémentaire	Passe tous les tests et le test supplémentaire	10/10
Critères de comparaison		
Lisibilité	Bonne lisibilité	5/5
Qualité	Bonne Qualité	4,8/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère unRLE : O(n) parcourt chaque paire de caractères	4,5/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 13,6789ms unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 10,9682ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça met 639ms	1,9/5
TOTAL Notation avec le barème donné		20/20
TOTAL Critère de comparaison		16,2/20
MOYENNE		18,1/20

2.2. 20simplicite.py

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat Interprète et une partie des algorithmes fonctionnent, RLE avec et sans itération ne marchent pas en raison d'une mauvaise indentation qui provoque une boucle while infinie	5/10
Tests JUnit + Test supplémentaire	Passe qu'une partie des tests : unRLE avec et sans itération et ne passe pas le test supplémentaire	4/10
Critères de comparaison		
Lisibilité	Peu lisible : présence de commentaires, mais noms de variables pas très explicites, et mauvaise indentation	1,9/5
Qualité	Petites erreurs quant aux espaces, et le bloc de RLE dupliqué pour RLERécursif	2,8/5
Efficacité	Complexité Algorithmique : RLE : / unRLE : O(n) parcourt la chaîne compressée	2/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met ne marche pas unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 0,030ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça ne marche pas	1,3/5
TOTAL Notation avec le barème donné		9/20
TOTAL Critère de comparaison		8/20
MOYENNE		8,5/20

2.3. 23simplicite.java

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat mais non-respect des consignes quant aux noms des méthodes (ClassicCompress et ClassicUnCompress) Compile, présence que des 2 méthodes sans itérations et l'algorithme RLE fonctionne dans le sens inverse : lettres avant les chiffres (abc -> a1b1c1) et le unRLE ne fonctionne pas (2a1b9c1c -> 22222222221111111111199999999999111111111111)	2,5/10
Tests JUnit + Test supplémentaire	Ne passe aucuns tests et sûrement pas le test supplémentaire	0/10
Critères de comparaison		
Lisibilité	Bonne lisibilité : présence de commentaires mais manque d'aération	4,4/5
Qualité	Bonne qualité sur les algorithme existants	4/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère unRLE : O(n) parcourt chaque paire de caractères	4,5/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 1,6522ms mais ça ne marche pas unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 10,346ms mais ça ne marche pas unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça ne marche pas	1,3/4
TOTAL Notation avec le barème donné		2,5/20
TOTAL Critère de comparaison		14,2/20
MOYENNE		8,34/20

2.4. 47simplicite.java

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat et respect des consignes quant aux méthode de java.util Compile et chaque algorithme fonctionne	10/10
Tests JUnit + Test supplémentaire	Passe tous les tests et le test supplémentaire	10/10
Critères de comparaison		
Lisibilité	Bonne lisibilité : noms de variables explicites et présence de commentaires	5/5
Qualité	Bonne qualité. Petites erreurs quant aux réaffectations des paramètres à l'intérieur des méthodes qui seraient à éviter	4/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère unRLE : O(n) parcourt chaque paire de caractères	4,5/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 15,1959ms unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 10,2178ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça met 524ms	1,9/5
TOTAL Notation avec le barème donné		20/20
TOTAL Critère de comparaison		15,4/20
MOYENNE		17,7/20

2.5. 61simplicite.c

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat et respect des consignes quant aux méthode de java.util Compile et chaque algorithme fonctionne	10/10
Tests JUnit + Test supplémentaire	Passe une partie des tests, pas RLE avec et sans itération car pas de vérification si la chaîne est vite ; passe le test supplémentaire	5/10
Critères de comparaison		
Lisibilité	Bonne lisibilité : noms de variables explicites et présence de commentaires	5/5
Qualité	Fuites de mémoire potentielle de certains algorithmes ; Problèmes quant à l'utilisation de sprintf (non vérification de tampon) ; Non-termination de la chaîne ('\0')	3,4/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère de la chaîne unRLE : O(n) parcourt la chaîne compressée	4,5/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 0,007ms unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 0,001ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça met 27,742ms	5/5
TOTAL Notation avec le barème donné		15/20
TOTAL Critère de comparaison		17,9/20
MOYENNE		16,45/20

2.6. Classement

CLASSEMENT SIMPLICITÉ	
Podium	Nom de l'algorithme
1er	**11simplicite.java** : Le code est excellent, les algorithmes passent tous les tests. Il a une bonne lisibilité, qualité et efficacité. Cependant il a un temps d'exécution assez long
2ème	**47simplicite.java** : Très bon code, chaque algorithme passe tous les tests. Très bonne lisibilité et efficacité. Cependant il y a quelques erreurs quant à la qualité et le temps d'exécution est assez long
3ème	**61simplicite.c** : Bon code, qui ne passe pas tous les tests en raison du fait qu'il n'y a pas de vérification quant au cas ou la chaîne en paramètre est vide, mais sinon tous les autres tests passent. La lisibilité et l'efficacité sont excellent, et le temps d'exécution est ultra rapide. Cependant il y a quelques erreurs quant à la qualité du code
4ème	**20simplicite.py** : Mauvais code, mauvaise indentation qui mène à des boucles infinies pour les algorithmes RLE avec et sans itérations, et bien sûr il ne passe pas tous les tests. Algorithmes peu lisibles, avec des erreurs quant à la qualité du code. Le temps pour unRLE est rapide, et l'efficacité est bonne pour cet algorithme
5ème	**23simplicite.java** : Mauvais code, non-respect des noms de méthodes imposés (ClassicCompress et ClassicUnCompress). Présence que de RLE et unRLE, et ils ne fonctionnent pas (pas de retours conformes). Aucun test n'est passé. Cependant bonne complexité quant au algorithmes existants, bonne lisibilité et qualité. Temps rapide pour RLE (mais ça ne marche pas) et les autres temps ne sont pas terribles ou pas récupérables

3. Sobriété

3.1. 27sobriete.java

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat et respect des consignes quant aux méthode de java.util Compile et chaque algorithme fonctionne	10/10
Tests JUnit + Test supplémentaire	Passe tous les tests et le test supplémentaire	10/10
Critères de comparaison		
Lisibilité	Bonne lisibilité : noms de variables explicites et présence de quelques commentaires	5/5
Qualité	Bonne qualité	4,4/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère unRLE : O(n) parcourt la chaîne compressée	4,5/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 12,242ms unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 13,9498ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça met 732ms	1,9/5
TOTAL Notation avec le barème donné		20/20
TOTAL Critère de comparaison		15,8/20
MOYENNE		17,9/20

3.2. 28sobriete.c

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat et respect des consignes quant aux méthode de java.util Compile, absence des algorithmes quant à la décompression. Échec de fonctionnement.	2/10
Tests JUnit + Test supplémentaire	Passe aucun test et surtout pas le test supplémentaire	0/10
Critères de comparaison		
Lisibilité	Assez lisible : présence de trop de commentaires qui rendent le fichier brouillon, mais noms de variables explicites	2,5/5
Qualité	Absence de vérification après les appels de 'malloc' et de 'realloc' ; Problème quant à l'utilisation de sprintf (non vérification de tampon) ; Non-termination de la chaîne ('\0') ; Duplication de code ; et d'autres problèmes mineurs	1,7/5
Efficacité	Complexité Algorithmique : RLE : O(n) algorithme parcourt chaque caractère unRLE : /	2/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça ne marche pas unRLE (in) : Pour la chaîne 2a1b9c1c, ça ne marche pas unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça met ne marche pas	0/4
TOTAL Notation avec le barème donné		2/20
TOTAL Critère de comparaison		6,2/20
MOYENNE		4,1/20

3.3. 51sobriete.java

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat et respect des consignes quant aux méthode de java.util Compile mais les RLE ne fonctionne pas très bien car ils ne respectent pas la règle : WWWWWWWWWW = 9W1W mais ça donne 10W	6/10
Tests JUnit + Test supplémentaire	Passe les 3/4 des tests, sauf les tests sur RLE sans itérations et ne passe pas le test supplémentaire	5,5/10
Critères de comparaison		
Lisibilité	Bonne lisibilité : noms de variables explicites	5/5
Qualité	Bonne qualité. Petites erreurs quant aux réaffectations des paramètres à l'intérieur des méthodes qui seraient à éviter	4,4/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère unRLE : O(n) parcourt la chaîne compressée	4,5/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 0,8403ms mais ça ne marche pas -> 2a1b10c unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 0,9334ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça ne marche pas	1,4/5
TOTAL Notation avec le barème donné		11,5/20
TOTAL Critère de comparaison		15,3/20
MOYENNE		13,4/20

3.4. 62sobriete.java

Notation des Algorithme avec le Barème donné		
Tests et Fonctionnement	Description des résultats	Note
Fonctionnement	Respect de l'anonymat et respect des consignes quant aux méthode de java.util Compile mais dans l'algorithme RLE (in,iteration) la personne fait appel à son algorithme RLE de son fichier Efficacite.java	7/10
Tests JUnit + Test supplémentaire	Passe 3/4 des tests mais passe le test supplémentaire (si on prend en compte qu'il a fait son algo d'efficacité)	6/10
Critères de comparaison		
Lisibilité	Bonne lisibilité : noms de variables explicites et présence de commentaires	5/5
Qualité	Bonne qualité	4,4/5
Efficacité	Complexité Algorithmique : RLE : O(n) L'algorithme parcourt chaque caractère unRLE : O(n) parcourt la chaîne compressée	4,5/5
Temps d'exécution	RLE (in) : Pour la chaîne aabccccccccc, ça met 18,1754ms unRLE (in) : Pour la chaîne 2a1b9c1c, ça met 17,2958ms unRLE (RLE(in,iteration),iteration) : Pour la chaîne wwwgzzzzzzzzz, ça met 429ms	1,6/5
TOTAL Notation avec le barème donné		13/20
TOTAL Critère de comparaison		15,5/20
MOYENNE		14,25/20

3.5. Classement

CLASSEMENT SOBRIÉTÉ	
Podium	Nom de l'algorithme
1er	**27sobriete.java** : Très bon code, les tests sont tous passés, la lisibilité, l'efficacité et la qualité sont très bons. Par contre le temps d'exécution est très long
2ème	**62sobriete.java** : Bon code, mais pour son RLE avec itération, il fait appel à l'algorithme d'une autre classe que je n'ai pas. Il passe les 3/4 des tests si on prend en compte qu'il a fait son algorithme. Très bonne lisibilité et efficacité et bonne qualité. Par contre les temps d'exécution est vraiment long par rapport au 27sobriete.java
3ème	**51sobriete.java** : Code Passable, mais problème avec RLE qui ne respecte pas la notation (pas le droit aux nombres, que des chiffres). Il passe les 3/4 des tests mais pas le test supplémentaire. Bonne lisibilité et efficacité, mais revoir les petites erreurs quant à la qualité. Temps d'exécution rapide pour les temps récupérés et pour les algorithmes qui fonctionnent
4ème	**28sobriete.java** : Très mauvais code, absence des algorithmes de décompression et échec de ceux existants. Ne passe aucun test. Trop de commentaires qui altèrent la lisibilité, mauvaise qualité et bonne qualité pour l'algorithme existant. Aucun temps d'exécution récupérable