SAE S2.02 : Exploration algorithmique d'un problème.

Argumentaire et méthodes utilisées :

Catégorie **simplicité** :

Pour cette catégorie, je me suis basé sur la lisibilité du code et sa compréhension, ainsi que sur la qualité du code (Codacy).

Simplicité - 159.py :

```
def bonmot(var):

   if var ==" " or var=="":
       return ("")
   space=0
   for i in var:
       if i == " ":
           space+=1
   if space==len(var):
       return(var)
   else:
       o=0
       if var[0] == " " and var[1] != " ":
            var = var[:0] + "" + var[1:]

       if var[-1] == " " and var[-2] != " ":
           var = var[:-1] + ""

       fin=0
       for i in range(-1,-len(var),-1):
          if var[i]==" ":
              fin +=1
          else:
            break

       print(fin)
       if (fin%2 ==0):
           o=0
       else:
           o=1
       NvMot=""
       strlen = len(var)
       i=0
       while i<strlen-o:
           if var[i]==" ":
```

```
            if var[i+1] ==" " or var[i-1]==" ":
                    NvMot+= var[i]
                    NvMot+= var[i+1]
                    i+=1
            else:
                NvMot += var[i]
            i+=1
        if (o>0):
           NvMot+=" "
        return (NvMot)
```

On peut voir que le code est très long pour un algorithme dit « simple » (environ 50 lignes), et il n'y a aucun commentaire, ce qui rend difficile sa compréhension. Néanmoins, les noms des variables sont très cohérents et donc intuitifs. Il y énormément de boucles et de conditions, ce qui fait tâche dans une catégorie de **simplicité**.

simplicite-159.py

Pour ce qui est de la qualité du code, Codacy l'a classé en rang **A**.

Simplicité - 2.java :

```java
package eraser;

public class Eraser {
    public static String erase(String str) {
        // Remplace tous les blancs précédés ou suivis de quoi que ce soit d'autre
        str = str.replaceAll("(\\S)\\s(\\S)", "$1$2");
        // Cas particulier du " "
        if (str.equals(" "))
            return "";
        if (str.length() > 1) {
            // Enlève un potentiel 1er espace (si au moins 1 caractère)
            if ((str.charAt(0) == ' ') && (str.charAt(1) != ' '))
                str = str.substring(1);
            if (str.length() > 1)
                // Enlève un potentiel dernier espace (si au moins 2 caractères)
                if ((str.charAt(str.length()-1) == ' ') && (str.charAt(str.length()-2) != ' ') )
                    str = str.substring(0, str.length()-1);
        }
        return(str);
    }
}
```

Ce code est court (20 lignes) et contient des commentaires pour aider à la compréhension. Chaque conditions ‘if’ est précédé d’u commentaire, qui explique les conditions pour rentrer dans cette dernière. Je trouve cette approche vraiment utile, et me permet de mieux comprendre le fonctionnement du code.

simplicite-2.java

Néanmoins, ce code est celui de plus basse qualité.

Simplicité - 26.java :

```java
import java.util.Scanner;

class RemoveSpaces{

  public static void main(String[] args) {
    String str = removeSpaces();
    System.out.println("\n" + str); // Appel à la fonction "removeSpaces"
  }

  public static String removeSpaces() {

    System.out.println("\nEntrez une chaine de caractères :");
    Scanner scanner = new Scanner(System.in);
    String str = scanner.nextLine();
    String tmp = ""; // Variable temporaire ou on stocke la chaine de caractère entrée
    int espace = 0; // Compteur d'espaces consécutifs

    for (int i = 0; i < str.trim().length(); i++) {
      int j = i;
      while (str.charAt(j) == ' ') {
        espace += 1;
        j += 1;
      }
      if (espace == 1) {
        if (str.charAt(i - 1) == ' ') {
          tmp += str.charAt(i);
        }
      } else {
        tmp += str.charAt(i);
      }
      espace = 0;
    }
    return tmp;
  }
}
```

Cet algorithme est de longueur moyenne (35 lignes), et comporte un peu de commentaires. Néanmoins, ce code ne respecte pas la structure déjà établi dans le répository, car le Main et le code sont présent sur le même fichier. De plus, le programme ne prend pas de paramètres, alors que les fichiers de test ont été établi pour appeler la fonction avec une chaine de caractères en entrée.

simplicite-26.java

Cet algorithme est de bonne qualité, d'après Codacy.

Simplicité - 47.c :

```c
char* erase(char* chaine) {
  int size = strlen(chaine);
  char resultat[size];
  int j = 0;
  for (int i = 0; i < size; i++) {
    if (chaine[i] == ' ') {
      if ((i+1 < size && chaine[i+1] == ' ') || (i !=0 && chaine[i-1] == ' ')){
        resultat[j]=chaine[i];
        j++;
      }
    }
    else
    {
      resultat[j] = chaine[i];
      j++;
    }
  }
  return resultat;
}
```

Ce dernier code est court (19 lignes) et ne comporte qu'une seule boucle et assez peu de conditions. Néanmoins, quelques commentaires pour comprendre les conditions n'auraient pas été de refus.

simplicite-47.c

Cet algorithme est de bonne qualité, d'après Codacy.

Classement final pour la catégorie Simplicité :

1er → simplicité-47.c	5pts
2ème → simplicité-2.java	4pts
3ème → simplicité-26.java	2pts (2-1, ne respecte pas la structure fournie)
4ème → simplicité-159.py	2pts

BIRET-TOSCANO Esteban 4A

Catégorie **efficacité** :

Pour établir le classement, j'ai mesuré le temps d'exécution des différents programmes, pour une même chaine de caractères.

Pour les programmes python, j'ai créé une boucle for qui tourne 1 000 000 fois, pour ensuite faire une moyenne de temps d'exécution sur tous ces essais et un écart type. J'ai dû importer les modules time et statistics.

```python
import time
import statistics

measures = []
for i in range(1000000):
    start = time.time()

    erase("bonjou r")

    end = time.time()
    measures.append(end - start)

mean = statistics.mean(measures)
stdev = statistics.stdev(measures)

print('Temps d\'exécution :')
print(f' - Moyenne : {mean:.1}ms')
print(f' - Écart-type : {stdev:.1}ms')
```

Chaine de caractères (courte) utilisée pour ces tests → bonjou r

Voyons ce que donne les tests sur les 3 programmes python :

Efficacité-96.py :

```python
# return string without spaces
def erase(cc):
  nouvelle_chaine=""
  taille = len(cc)


  if(taille>0):
    #premier caractère (si ce n'est pas un espace, ou qu'il y a un espace après, alors on recopie)
    if(cc[0]!=" " or cc[1]==" "):
        nouvelle_chaine+=cc[0]

    #tous les caractères sauf premier et dernier (si ce n'est pas un espace, ou qu'il y a un espace avant ou après, alors on recopie)
    for i in range (1, taille-1):
        if(cc[i] != ' ' or cc[i - 1] == ' ' or cc[i + 1] == ' '):
            nouvelle_chaine+=cc[i]

    i=taille-1
    #dernier caractère (si ce n'est pas un espace, ou qu'il y a un espace avant, alors on recopie)
    if(cc[i] != ' ' or cc[i - 1] == ' '):
       nouvelle_chaine+=cc[taille-1]
  return nouvelle_chaine
```

```
Temps d'exécution :
 - Moyenne : 2e-06ms
 - Écart-type : 3e-06ms
```

Efficacité-46.py :

```python
# return string without spaces
def erase2(cc):
    chaine = ""
    for i in range(len(cc)):
        if i == len(cc) - 1 and cc[i] == " ":
            if cc[i] == " ":
                chaine += ""
            else:
                chaine += cc[-1]
            break

        if cc[i] != " ":
            chaine += cc[i]
        elif (cc[i] == " " and cc[i + 1]) == " " or (cc[i] == " " and cc[i - 1] == " "):
            chaine += cc[i]
        else:
            chaine += ""
    return chaine
```

```
Temps d'exécution :
 - Moyenne : 3e-06ms
 - Écart-type : 8e-07ms
```

Efficacité-126.py :

```python
# return string without spaces
def erase(cc):
    re.sub("([^\s])\s([^\s])", r"\1\2", re.sub("([^\s])\s([^\s])", r"\1\2", cc).strip())

    pass
```

```
Temps d'exécution :
 - Moyenne : 7e-06ms
 - Écart-type : 3e-06ms
```

A noter que j'ai dû rajouter une ligne dans ce programme pour qu'il fonctionne → malus de 1 point.

import re

J’ai suivi la même démarche pour le programme Java.

```
		long startTime = System.currentTimeMillis();
		for (int i=0; i < 1000000; ++i) {
			erase("bonjou  r");
		}
		long endTime = System.currentTimeMillis();
		System.out.println("Moyenne du temps d'exécution : "+
((double)(endTime-startTime)/1000000) + "ms");
```

Ce qui nous donne pour le programme Efficacité - 24.java :

```
package main.java.eraser;

public class Eraser {

	/***
	 * Méthode permettant de supprimer les espaces simples et gardant les espaces multiples
	 * @param str
	 * @return str
	 */
    public static String erase(String str) {
        int tailleM ;
        String newStr = "";
        int cmpt = 0;
        tailleM = str.length() ;

        for (int i = 0 ; i < tailleM ; i ++) {
                char caract = str.charAt(i);
                if (caract == ' ') {
                cmpt ++; // Compte du nombre d'espace
            }
                else {
                        if (cmpt > 1) { // Si le nombre  d'espace est suppérieur à 1 alors on les affiches
                                for (int j = 0; j < cmpt ; j++) {
                                        newStr += ' ';
                                        }
                }
                        cmpt = 0;
                        newStr += caract ;
                        }
        }

        return newStr;
    }
}
```

```
Moyenne du temps d'exécution : 4.7E-4ms
```

Pour une chaine de caractères **courte**, nous obtenons ce classement, du temps d'exécution du plus court au plus long →

Efficacité – 24.java : 4.7e – 4 ms → environ 0.00047 millisecondes.

Efficacité – 96.py : 2e – 06 ms → environ 0.2 millisecondes.

Efficacité – 46.py : 3e – 06 ms → environ 0.3 millisecondes.

Efficacité – 126.py : 7e – 06 ms → environ 0.7 millisecondes.

Refaisons le même test, mais cette fois ci avec une chaine de caractère **plus longue** :

Pour Python :

```python
import time
import statistics

measures = []
for i in range(1000000):
    start = time.time()

    erase("Go Out On a Limb  Meaning: Putti  ng yourse    lf in a risky situat   ion in order to help someon    e; or to h

    end = time.time()
    measures.append(end - start)

mean = statistics.mean(measures)
stdev = statistics.stdev(measures)

print('Temps d\'exécution :')
print(f' - Moyenne : {mean:.1}ms')
print(f' - Écart-type : {stdev:.1}ms')
```

.....

Pour Java :

```java
public static void main(String[] args) {

            long startTime = System.currentTimeMillis();
            for (int i=0; i < 1000000; ++i) {
                  erase("Go Out On a Limb  Meaning: Putti  ng yourse    lf in a
risky situat   ion in order to help someon    e; or to hazard a guess.   A Little
from Colum      n A, a Little from Column BMeaniqff    ng: A course of action draw
ing a couple of different factors reasons.");
            }
            long endTime = System.currentTimeMillis();
            System.out.println("Moyenne du temps d'exécution : "+
((double)(endTime-startTime)/1000000) + "ms");


      }
```

La chaine de caractère longue pour ce test →

Go Out On a Limb Meaning: Putti ng yourse   lf in a risky situat   ion in order to help someon   e; or to hazard a guess.   A Little from Colum   n A, a Little from Column BMeaniqff   ng: A course ofaction draw   ing a couple of different factors reasons.

Pour une chaine de caractères **longue**, nous obtenons ce classement, du temps d'exécution du plus court au plus long →

Efficacité – 24.java : `0.008186` millisecondes.

Efficacité – 96.py : 4e – 05 ms → environ 0.4 millisecondes.

Efficacité – 46.py : 8e – 05 ms → environ 0.8 millisecondes.

Efficacité – 126.py : 8e – 05 ms → environ 0.8 millisecondes.

Nous voyons que le programme 46.py est beaucoup moins efficace quand le nombre de caractères croît, alors que le 126 n'augmente pas tellement. Quant à lui, le programme 24.java est toujours très rapide. Le programme 96 a doublé son temps d'exécution.

Classement final pour la catégorie Efficacité :

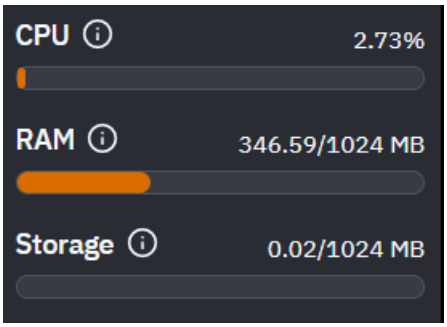
1er → Efficacité – 24.java	5pts
2ème → Efficacité – 96.py	4pts
3ème → Efficacité – 46.py	3pts
4ème → Efficacité – 126.py	1pt (2-1, malus pour l'import manquant)

BIRET-TOSCANO Esteban 4A

Catégorie **sobriété numérique** :

Pour établir le classement, j'ai utilisé les informations que fournissait Replit au niveau du CPU, de la RAM et de l'espace de stockage utilisé par le programme. J'ai bien évidemment fait les tests avec la même chaine de caractère pour tous les algos. **(C ou co u J M B).**

Sobriété-113.java :



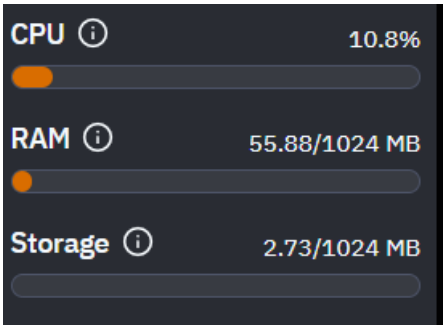
CPU = La puissance de traitement utilisée.

RAM = La mémoire utilisée.

Storage = L'espace de stockage utilisé.

Sobriété-133.c : Ce programme ne fonctionnant pas, je lui ai mis la note de 1 point.

Sobriété-76.c :



Classement final pour la catégorie Sobriété Numérique :

1er → Sobriété – 76.c	5pts
2ème → Sobriété – 113.java	4pts
3ème → Sobriété – 133.c	1pt (ne fonctionne pas pour toutes les chaines de caractères)