Yann Guillevic 4B – B.U.T. Informatique – 2022

Compte Rendu SAE 2.02

Évaluation des algorithmes

Tableau récapitulatif :

Code couleur :

: algorithme efficacité

: algorithme simplicité

: algorithme sobriété

Pour le classement par catégorie, j'ai privilégié les algos qui réussissent tous les tests correctement.

Classement : le meilleur à 4, le pire à 1

Les notes sur 5 ne sont pas significatives. Elles permettent juste de voir quel algorithme est meilleur qu'un autre dans un critère.

Classement par catégorie :

Numéro algo	Lisibilité du code	Qualité du code	Efficacité	Sobriété numérique	Temps d'exécution	Test	Classement
4	4.5/5	5/5	3/5	3/5	5/5	oui	3
6	5/5	5/5	3/5	5/5	2.5/5	non	2
23	5/5	4/5	5/5	3/5	3/5	non	1
112	4.5/5	5/5	4/5	3/5	4/5	oui	4
42	4.75/5	5/5	5/5	3/5	5/5	oui	4
61	5/5	2.5/5	5/5	5/5	2/5	non	1
79	4,25/5	5/5	4/5	4/5	3/5	oui	3
88	4.5/5	5/5	5/5	2/5	4/5	non	2
136	4,25/5	5/5	4/5	4/5	5/5	oui	4
145	5/5	5/5	4/5	5/5	4/5	non	3
161	1/5	2/5	4/5	3/5	3/5	oui	2

Pour l'algo efficacité numéro 4 :

Lisibilité du code :
La lisibilité est claire, le code n'est pas très long et les boucles sont bien formées. Le seul petit point négatif est le manque de javadoc mais dans l'ensemble, l'algorithme est clair.

Qualité du code :
En utilisant l'outil Pep8 Online, je constate que la qualité du code est excellente. Il n'y a aucune erreur importante renvoyé par le site et l'évaluation est parfaite.

Efficacité :
J'utilise un compteur qu'on incrémente de un chaque fois qu'on rentre dans une boucle for ou une boucle if :

Caractères en entrée	5	10	100	1000	5000
Compteur	10	20	200	2000	10000

On peut voir rapidement qu'il y a, pour un certains nombre de caractère, le double d'itération de boucle. Si on dit que le nombre de caractère est « n » alors sa complexité algorithmique vaut O(2n).

Tests :
L'algo passe correctement tous les tests.

Sobriété numérique :
Pour calculer la sobriété numérique des algorithmes Python, j'utilise la fonction « getllocatedblocks » du module « sys » qui renvoie le nombre de blocs mémoires alloués lors de l'exécution du programme.

Caractères en entrée	5	10	100	1000	5000
Nombre de blocs mémoires alloués	14	8	12	20	20

Temps d'exécution :

Nombre de donnée en entrée	5	10	100	200	400
Temps	41µs	44µs	73µs	102µs	199µs

Pour l'algo efficacité numéro 6 :

Lisibilité du code :

La lisibilité est claire et simple, la javadoc associée sur chaque ligne permet une grande compréhension de ce que fait l'algorithme. Le code reste d'une bonne longueur, il est aéré et les boucles sont simples.

Qualité du code :

Pour mesurer la qualité du code en java j'ai utilisé l'outil Codacy.
Ainsi, je peux dire que la qualité du code est très bonne. Il n'y a aucune erreur que ce soit dans la présentation ou dans l'écriture.

Grade	Filename	Issues	Complexity	Duplication
A	efficacite-6.java	0	-	0

Efficacité :

J'utilise un compteur qu'on incrémente de un chaque fois qu'on rentre dans une boucle for ou une boucle if :

Caractères en entrée	5	10	100	1000	5000
Compteur	8	18	198	1998	9998

On peut voir rapidement qu'il y a, pour un certains nombre de caractère, le double d'itération de boucle. Si on dit que le nombre de caractère est « n » alors sa complexité algorithmique vaut O(2n).

Tests :

L'algo ne passe pas tous les tests. Si il y a 2 espaces consécutifs au début du texte, l'algo en supprime 1.

Sobriété numérique :

L'utilisation mémoire avant la lancement du programme a été évaluée a 709 Mo.
Ce screen montre ainsi différentes valeurs sur l'utilisation mémoire de l'algorithme pour un certain nombre de caractères en entrée (avant lancement programme,5,10,100,1000) :

	Heure	Allocations (Diff.)	Taille du tas (Diff.)
1	9,59 s	26 (Non applicable)	7,13 Ko (Non applicable)
2	15,64 s	1 070 (+1 044 ↑)	1 376,64 Ko (+1 369,51 Ko ↑)
3	24,58 s	4 464 (+3 394 ↑)	597,94 Ko (-778,69 Ko ↓)
4	34,29 s	13 358 (+8 894 ↑)	1 428,42 Ko (+830,48 Ko ↑)
5	45,15 s	14 869 (+1 511 ↑)	1 510,32 Ko (+81,90 Ko ↑)

L'utilisation maximum de la mémoire a été de 755 Mo.

Temps d'exécution :

Nombre de donnée en entrée	5	10	100	200	400
Temps	115µs	123µs	450µs	950µs	1,5ms

Pour l'algo efficacité numéro 23 :

Lisibilité du code :
La lisibilité est claire et simple, la javadoc associée sur chaque ligne permet une grande compréhension de ce que fait l'algorithme. Le code est aéré et les boucles sont simples.

Qualité du code :
Pour mesurer la qualité du code en python j'ai utilisé l'outil PEP8 Online :http://pep8online.com/checkresult
Ainsi, je peux dire que la qualité du code est très bonne. L'outil ne renvoie que 2 inconvénients majeurs qui sont :

- un espace entre le « list » et la parenthèse :

```
l = list (texte)
```

- le fait de coller l'opérateur avec la valeur :

```
i+=1
```

Efficacité :
J'utilise un compteur qu'on incrémente de un chaque fois qu'on rentre dans une boucle for ou une boucle if :

Caractères en entrée	5	10	100	1000	5000
Compteur	3	8	98	998	4998

On remarque que le nombre de caractère en entrée correspond au nombre de tour de boucle que fait l'algo. Si « n » correspond au nombre de caractère en entrée, alors on peut dire que cet algo est de complexité O(n).

Tests :
L'algo ne passe pas correctement tous les tests. Si la chaîne en entrée ne contient aucun caractère, une erreur apparaît dans le terminal.

```
IndexError                                Traceback (most recent call last)
<ipython-input-313-db5bcfe64079> in <module>
     11
     12 # Traitement spécifique pour supprimer un éventuel espace en tête
---> 13 if l[0] == ' ' and l[1] != ' ':
     14     del(l[0])
     15

IndexError: list index out of range
```

Sobriété numérique :
Pour calculer la sobriété numérique des algorithmes Python, j'utilise la fonction « getllocatedblocks » du module « sys » qui renvoie le nombre de blocs mémoires alloués lors de l'exécution du programme.

Caractères en entrée	5	10	100	1000	5000
Nombre de blocs mémoires alloués	4	10	9	10	10

Temps d'exécution :

Nombre de donnée en entrée	5	10	100	200	400
Temps	500µs	530µs	547µs	590µs	650µs

Pour l'algo efficacité numéro 112 :

Lisibilité du code :

La lisibilité est assez bonne, le code est simple et aéré. Le seul petit inconvénient est le fait qu'il n'y a pas de javadoc pour expliquer le code mais le programme est suffisamment bien écrit pour le comprendre assez clairement.

Qualité du code :

Pour mesurer la qualité du code en java j'ai utilisé l'outil Codacy.
Ainsi, je peux dire que la qualité du code est très bonne. Il n'y a aucune erreur que ce soit dans la présentation ou dans l'écriture.

Efficacité :

J'utilise un compteur qu'on incrémente de un chaque fois qu'on rentre dans une boucle for ou une boucle if :

Caractères en entrée	5	10	100	1000	5000
Compteur	9	18	180	1800	9000

Grâce à ce tableau, on peut apercevoir plusieurs choses sur la complexité algorithmique de cet algo. D'abord, quand on multiplie par 2 le nombre de caractère en entrée, le nombre d'itération est lui aussi multiplié par 2. La même chose se reproduit lorsqu'on multiplie par 10.
Si on se base sur les temps d'exécution, on remarque que même en augmentant la taille d'entrée par 2 ou plus, le temps d'exécution reste quasi-linéaire.

Ainsi, on peut dire que la complexité algorithmique de cet algo est de O(nlog(n)) qui est un tout petit peu moins bien que O(n).

Tests :

L'algo passe bien tous les tests.

Sobriété numérique :

L'utilisation mémoire avant la lancement du programme a été évaluée a 709 Mo.
Ce screen montre ainsi différentes valeurs sur l'utilisation mémoire de l'algorithme pour un certain nombre de caractères en entrée (avant lancement programme,5,10,100,1000) :

	Heure	Allocations (Diff.)	Taille du tas (Diff.)
		Profilage du tas natif activé. Allocations précédentes non i	
1	12,97 s	9 (Non applicable)	0,07 Ko (Non applicable)
2	20,24 s	2 487 (+2 478 ↑)	27 107,63 Ko (+27 107,56 Ko ↑)
3	28,18 s	3 210 (+723 ↑)	17 140,74 Ko (-9 966,89 Ko ↓)
4	37,18 s	4 464 (+1 254 ↑)	26 283,33 Ko (+9 142,59 Ko ↑)
5	49,15 s	4 603 (+139 ↑)	4 291,45 Ko (-21 991,89 Ko ↓)

L'utilisation maximum de la mémoire a été de 771 Mo.

Temps d'exécution :

Nombre de donnée en entrée	5	10	100	200	400
Temps	65µs	71µs	130µs	190µs	300µs

Pour l'algo simplicité numéro 42 :

Lisibilité du code :
La lisibilité est claire et simple, la javadoc associée sur chaque ligne permet une grande compréhension de ce que fait l'algorithme. Le code est aéré. Les boucles « if » sont un peu chargées mais c'est le seul inconvénient qu'on peut retenir de ce code si on veux vraiment aller loin.

Qualité du code :
Pour mesurer la qualité du code en java j'ai utilisé l'outil Codacy.
Ainsi, je peux dire que la qualité du code est très bonne. Il n'y a aucune erreur que ce soit dans la présentation ou dans l'écriture.

Grade	Filename	Issues	Complexity	Duplication
A	simplicite-42.java	0	-	0

Efficacité :
J'utilise un compteur qu'on incrémente de un chaque fois qu'on rentre dans une boucle for ou une boucle if :

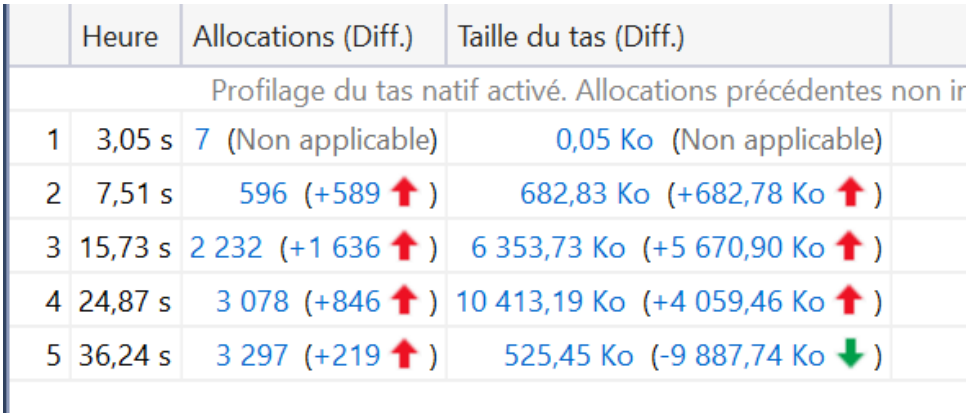
Caractères en entrée	5	10	100	1000	5000
Compteur	5	10	100	1000	5000

Grâce au tableau on remarque rapidement que le nombre de tour de boucle correspond au nombre de caractère en entrée.

Ainsi, on peut dire que la complexité algorithmique de cet algo est de O(n).

Tests :
L'algorithme passe bien tous les tests.

Sobriété numérique :
Pour évaluer la sobriété numérique de cet algo, j'ai utilisé l'outil Microsoft Visual Studio (version 2015) qui permet de retourner l'utilisation mémoire d'un programme en exécution.
Ce screen montre ainsi différentes valeurs sur l'utilisation mémoire de l'algorithme pour un certain nombre de caractères en entrée (avant lancement programme,5,10,100,1000) :



	Heure	Allocations (Diff.)	Taille du tas (Diff.)
		Profilage du tas natif activé. Allocations précédentes non ir	
1	3,05 s	7 (Non applicable)	0,05 Ko (Non applicable)
2	7,51 s	596 (+589 ↑)	682,83 Ko (+682,78 Ko ↑)
3	15,73 s	2 232 (+1 636 ↑)	6 353,73 Ko (+5 670,90 Ko ↑)
4	24,87 s	3 078 (+846 ↑)	10 413,19 Ko (+4 059,46 Ko ↑)
5	36,24 s	3 297 (+219 ↑)	525,45 Ko (-9 887,74 Ko ↓)

L'utilisation maximum de la mémoire a été de 761 Mo.

Temps d'exécution :

Nombre de donnée en entrée	5	10	100	200	400
Temps	77µs	83µs	220µs	300µs	430µs

Pour l'algo simplicité numéro 61 :

Lisibilité du code :

La lisibilité est claire et simple, la javadoc associée sur chaque ligne permet une grande compréhension de ce que fait l'algorithme. Le code est aéré et les boucles sont simples.

Qualité du code :

Pour mesurer la qualité du code en java j'ai utilisé l'outil Codacy.
D'après lui, cet algorithme ne présente aucune erreur.

Grade	Filename	Issues	Complexity	Duplication
A	simplicite-61.java	0	-	0

Cependant, l'algorithme renvoie une variable de type String qui n'est jamais initialisée. Donc l'algo ne peut jamais être exécuté puisque cette variable n'est jamais utilisée. Cela doit sans doute être une erreur d'inattention puisque le programme ne renvoie pas la bonne variable de retour.
(Pour réaliser les tests suivant, j'ai dû modifier le code pour changer cette variable de retour).

Efficacité :

J'utilise un compteur qu'on incrémente de un chaque fois qu'on rentre dans une boucle for ou une boucle if :

Caractères en entrée	5	10	100	1000	5000
Compteur	5	10	100	1000	5000

Grâce au tableau on remarque rapidement que le nombre de tour de boucle correspond au nombre de caractère en entrée.

Lorsqu'on regarde le tableau des temps d'exécution, on peut voir que lorsqu'on commence a mettre des grands nombres de données en entrée, le temps d'exécution a tendance a doublée quand on double les données en entrée . Donc on peut dire que la complexité algorithmique de cet algo est O(n).

Tests :
Le code a besoin d'être modifié pour pouvoir fonctionné. On peut dire qu'il ne respecte pas les tests.

Sobriété numérique :
L'utilisation mémoire avant la lancement du programme a été évaluée a 817 Mo.
Ce screen montre ainsi différentes valeurs sur l'utilisation mémoire de l'algorithme pour un certain nombre de caractères en entrée (avant lancement programme,5,10,100,1000) :

	Heure	Allocations (Diff.)	Taille du tas (Diff.)	
		Profilage du tas natif activé. Allocations précédentes non incluses.		
1	26,87 s	21 (Non applicable)	21,41 Ko (Non applicable)	
2	33,66 s	281 (+260 ↑)	230,91 Ko (+209,50 Ko ↑)	
3	44,06 s	966 (+685 ↑)	340,70 Ko (+109,79 Ko ↑)	
4	53,00 s	1 504 (+538 ↑)	5 163,63 Ko (+4 822,93 Ko ↑)	
5	62,03 s	4 942 (+3 438 ↑)	1 806,22 Ko (-3 357,40 Ko ↓)	

L'utilisation maximum de la mémoire a été de 822 Mo.

Temps d'exécution :

Nombre de donnée en entrée	5	10	100	200	400
Temps	120µs	150µs	800µs	1,3ms	2,6ms

Pour l'algo simplicité numéro 79 :

Lisibilité du code :

La lisibilité est claire et simple. L'algorithme est assez court et les boucles ne sont pas chargées. Il n'y a pas de javadoc pour expliquer les lignes mais le code est tout de même compréhensible.

Qualité du code :

Pour mesurer la qualité du code en java j'ai utilisé l'outil Codacy.
Ainsi, je peux dire que la qualité du code est très bonne. Il n'y a aucune erreur que ce soit dans la présentation ou dans l'écriture.

Grade	Filename	Issues	Complexity	Duplication
A	simplicite-79.java	0	-	0

Efficacité :

J'utilise un compteur qu'on incrémente de un chaque fois qu'on rentre dans une boucle for ou une boucle if :

Caractères en entrée	5	10	100	1000	5000
Compteur	11	22	220	2200	11000

Grâce au tableau on peut voir qu'il y a un peu plus de 2 fois le nombre d'itération de boucle par rapport au nombre de caractère en entrée

Ainsi, on peut dire que la complexité algorithmique de cet algo est de O(2n).

Tests :

Le code passe bien tous les tests.

Sobriété numérique :

L'utilisation mémoire avant la lancement du programme a été évaluée a 762 Mo.
Ce screen montre ainsi différentes valeurs sur l'utilisation mémoire de l'algorithme pour un certain nombre de caractères en entrée (avant lancement programme,5,10,100,1000) :

	Heure	Allocations (Diff.)	Taille du tas (Diff.)
	Profilage du tas natif activé à 12,98 s. Allocations précéde		
1	14,34 s	4 (Non applicable)	0,03 Ko (Non applicable)
2	19,39 s	122 (+118 ↑)	593,94 Ko (+593,91 Ko ↑)
3	27,34 s	1 018 (+896 ↑)	4 849,56 Ko (+4 255,62 Ko ↑)
4	36,95 s	1 193 (+175 ↑)	317,16 Ko (-4 532,40 Ko ↓)
5	49,51 s	1 667 (+474 ↑)	5 651,89 Ko (+5 334,73 Ko ↑)

L'utilisation maximum de la mémoire a été de 778 Mo.

Temps d'exécution :

Nombre de donnée en entrée	5	10	100	200	400
Temps	121µs	170µs	600µs	1,3ms	2,1ms

Pour l'algo simplicité numéro 88 :

Lisibilité du code :

La lisibilité est claire et simple. L'algorithme est assez court et les boucles ne sont pas chargées. Il n'y a pas de javadoc pour expliquer les lignes mais le code est tout de même compréhensible.

Qualité du code :

Pour mesurer la qualité du code en java j'ai utilisé l'outil Codacy.
Ainsi, je peux dire que la qualité du code est très bonne. Il n'y a aucune erreur que ce soit dans la présentation ou dans l'écriture.

Efficacité :

J'utilise un compteur qu'on incrémente de un chaque fois qu'on rentre dans une boucle for ou une boucle if :

Caractères en entrée	5	10	100	1000	5000
Compteur	4	9	99	999	4999

Grâce au tableau on remarque rapidement que le nombre de tour de boucle correspond au nombre de caractère en entrée.

Ainsi, on peut dire que la complexité algorithmique de cet algo est de O(n).

Tests :

Le code ne passe pas bien tous les tests. Si des espaces consécutifs se trouvent en début de chaîne, l'algorithme les supprime.

Sobriété numérique :

L'utilisation mémoire avant la lancement du programme a été évaluée a 688 Mo.
Ce screen montre ainsi différentes valeurs sur l'utilisation mémoire de l'algorithme pour un certain nombre de caractères en entrée (avant lancement programme,5,10,100,1000) :

Heure	Allocations (Diff.)	Taille du tas (Diff.)
5,66 s	193 (Non applicable)	213,57 Ko (Non applicable)
13,48 s	114 563 (+114 370 ↑)	21 059,78 Ko (+20 846,21 Ko ↑)
25,41 s	173 777 (+59 214 ↑)	21 993,89 Ko (+934,11 Ko ↑)
34,12 s	181 980 (+8 203 ↑)	29 517,61 Ko (+7 523,72 Ko ↑)
45,24 s	183 944 (+1 964 ↑)	32 822,66 Ko (+3 305,05 Ko ↑)
54,87 s	185 856 (+1 912 ↑)	32 088,98 Ko (-733,69 Ko ↓)

L'utilisation maximum de la mémoire a été de 744 Mo.

Temps d'exécution :

Nombre de donnée en entrée	5	10	100	200	400
Temps	440µs	450µs	700µs	850µs	920µs

Pour l'algo sobriété numéro 136 :

Lisibilité du code :
Je trouve l'algorithme assez complexe a comprendre mais cela peut se comprendre par mon bas niveau en C. Cependant le manque de javadoc y est certainement lié. La lisibilité est cependant claire et aéré. L'algorithme est assez court et les boucles ne sont pas longues.

Qualité du code :
Pour mesurer la qualité du code en java j'ai utilisé l'outil Codacy.
Ainsi, je peux dire que la qualité du code est très bonne. Il n'y a aucune erreur que ce soit dans la présentation ou dans l'écriture.

A	sobriete-136.c	0	-	0

Efficacité :
J'utilise un compteur qu'on incrémente de un chaque fois qu'on rentre dans une boucle for ou une boucle if :

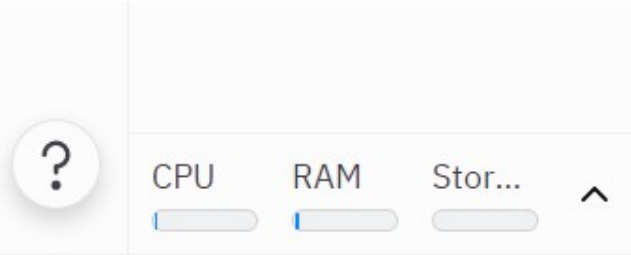
Caractères en entrée	5	10	100	1000	5000
Compteur	10	20	200	2000	10000

Grâce au tableau on peut voir qu'il y a un peu plus de 2 fois le nombre d'itération de boucle par rapport au nombre de caractère en entrée

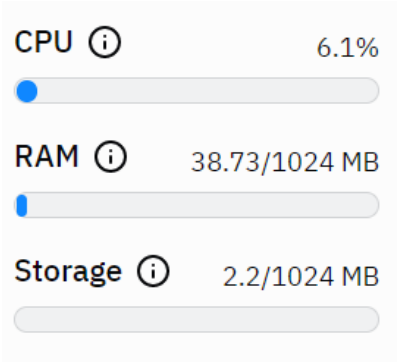
Cependant, lorsqu'on observe les temps d'exécution du programme, on remarque que les temps sont très linéaire. On pourrait donc dire que la complexité algorithmique de cet algo est de O(nlog(n)).

Compilation :
Le code passe bien tous les tests.

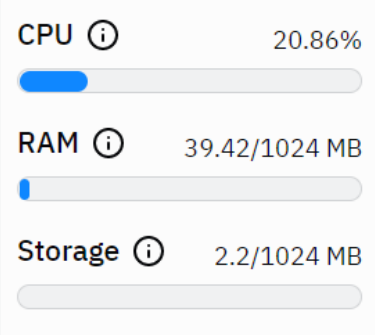
Sobriété numérique :
Pour évaluer la sobriété numérique d'un algorithme en C, j'utilise dans Replit l'onglet dédié lorsqu'on code un programme. (en bas à gauche de l'écran) :



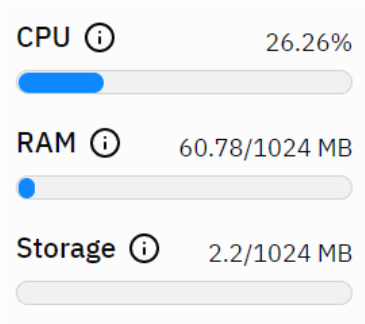
Avec une chaîne en entré de 5 caractères :



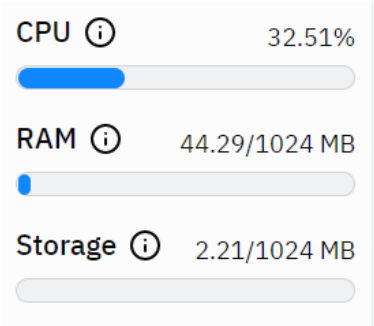
Avec une chaîne en entré de 10 caractères :



Avec une chaîne en entré de 100 caractères :



Avec une chaîne en entré de 1000 caractères :



Temps d'exécution :

Nombre de donnée en entrée	5	10	100	200	400
Temps	50µs	52µs	64µs	67µs	66µs

Pour l'algo numéro 145 :

Lisibilité du code :
La lisibilité est claire et simple. L'algorithme est assez court et les boucles ne sont pas chargées. La javadoc décrit bien se que les lignes font, se qui aide à la compréhension.

Qualité du code :
Pour mesurer la qualité du code en java j'ai utilisé l'outil Codacy.
Ainsi, je peux dire que la qualité du code est très bonne. Il n'y a aucune erreur que ce soit dans la présentation ou dans l'écriture.

Efficacité :
J'utilise un compteur qu'on incrémente de un chaque fois qu'on rentre dans une boucle for ou une boucle if :

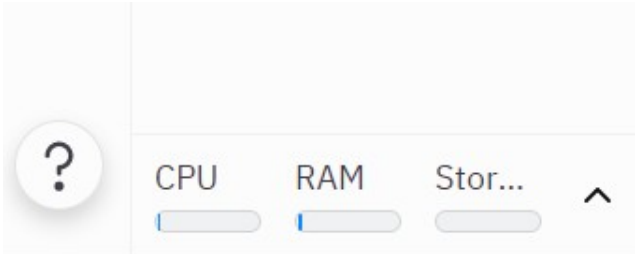
Caractères en entrée	5	10	100	1000	5000
Compteur	10	20	200	2000	10000

Grâce au tableau on peut voir qu'il y a un peu plus de 2 fois le nombre d'itération de boucle par rapport au nombre de caractère en entrée

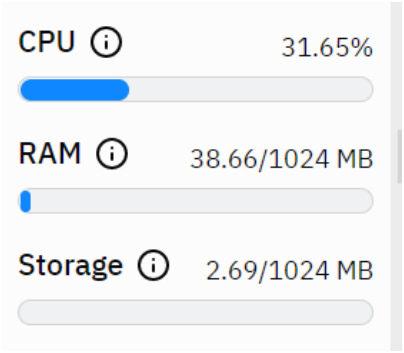
Cependant, lorsqu'on observe les temps d'exécution du programme, on remarque que les temps sont très linéaire. On pourrait donc dire que la complexité algorithmique de cet algo est de O(nlog(n)).

Compilation :
L'algorithme ne réalise aucun test correctement.

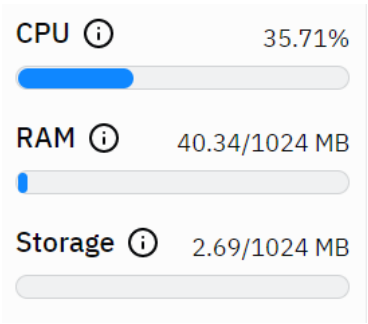
Sobriété numérique :
Pour évaluer la sobriété numérique d'un algorithme en C, j'utilise dans Replit l'onglet dédié lorsqu'on code un programme. (en bas à gauche de l'écran) :



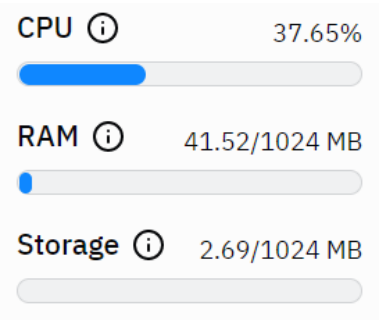
Avec une chaîne en entré de 5 caractères :



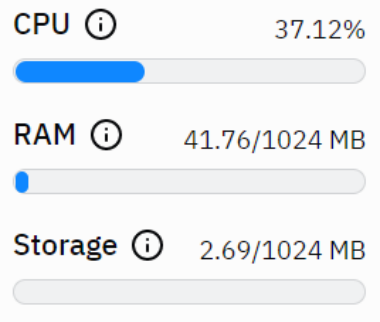
Avec une chaîne en entré de 10 caractères :



Avec une chaîne en entré de 100 caractères :



Avec une chaîne en entré de 1000 caractères :



Temps d'exécution :

Nombre de donnée en entrée	5	10	100	200	400
Temps	51µs	50µs	58µs	65µs	77µs

Pour l'algo numéro 161 :

Lisibilité du code :
La lisibilité est assez mauvaise. Il n'y a pas de javadoc pour expliquer ce que fait l'algo, les noms de variables ne sont pas explicité (exemeple : « o ») et l'algorithme est assez long avec des boucles récurrentes.

Qualité du code :
En utilisant pep8 Online, je peux dire que l'algorithme est de qualité moyenne L'outil que j'utilise me renvoi plus d'une cinquantaine de problème mais ce sont, en grande partie, des problèmes d'indentation ou d'espace entre opérateur.

Efficacité :
J'utilise un compteur qu'on incrémente de un chaque fois qu'on rentre dans une boucle for ou une boucle if :

Caractères en entrée	5	10	100	1000	5000
Compteur	20	36	324	3204	16004

Lorsqu'on regarde de ce tableau, on remarque que les compteurs sont trois fois plus grand que la chaîne de caractère qu'on met en entrée. On pourrait faire une première estimation de la complexité algorithmique à O(3n).
Si on regarde les temps d'exécution, on peut voir que le temps est assez linéaire au début mais moins quand on as une plus grosse valeur d'entrée. Ainsi, on peut dire que la complexité algorithmique de cet algo est O(nlog(n)).

Tests :
Le code passe bien tous les tests.

Sobriété numérique :
Pour calculer la sobriété numérique des algorithmes Python, j'utilise la fonction « getllocatedblocks » du module « sys » qui renvoie le nombre de blocs mémoires alloués lors de l'exécution du programme.

Caractères en entrée	5	10	100	1000	5000
Nombre de blocs mémoires alloués	50	60	60	65	62

Temps d'exécution :

Nombre de donnée en entrée	5	10	100	200	400
Temps	90µs	103µs	140µs	170µs	250µs