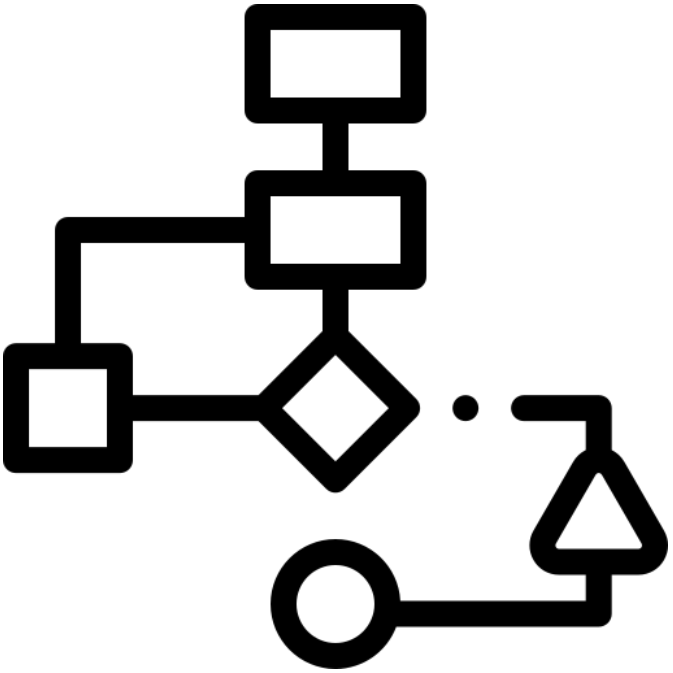
SAE 2.02 - Exploration algorithmique d'un problème



Loan Gayrard
1B

Table des matières

I - Algorithmes étudiés

Pour cette SAE, j'ai eu l'occasion d'étudier les algorithmes suivants :

Java :
efficacite-131.java
efficacite-75.java
simplicite-28.java
simplicite-54.java
simplicite-69.java
simplicite-82.java
sobriete-92.java

Python :
efficacite-4.py
efficacite-134.py
sobriete-102.py

C :
sobriete-76.c

II - Tests des algorithmes

Une fois les algorithmes récupérés, j'ai tout d'abord effectué des tests de validité de ces derniers. Ces tests comportent :

- Tests de compilation / éxécution
- Test de validité (est-ce qu'ils remplissent la tâche demandée ?)

1) Tests de compilation / éxécution

Java :
efficacite-131.java -> Erreur de compilation (Duplicate local variable i)
efficacite-75.java -> OK
simplicite-28.java -> OK
simplicite-54.java -> Erreur de compilation (Unreachable code)
simplicite-69.java -> OK
simplicite-82.java -> OK
sobriete-92.java -> OK

Python :
efficacite-4.py -> OK
efficacite-134.py -> OK
sobriete-102.py -> OK

C :
sobriete-76.c -> OK

2) Tests de validité

Phrase en entrée : “ T e s t e r aser ”
Phrase attendue en sortie : “Test eraser”

Java :
efficacite-131.java “ T e s t e r aser ” -> Erreur de compilation (Duplicate local variable i)
efficacite-75.java : “ T e s t e r aser ” -> "Test eraser" OK
simplicite-28.java : “ T e s t e r aser ” -> "Test eraser" OK
simplicite-54.java -> Erreur de compilation (Unreachable code)
simplicite-69.java : “ T e s t e r aser ” -> "Test eraser" OK
simplicite-82.java : “ T e s t e r aser ” -> "Test eraser" OK
sobriete-92.java : “ T e s t e r aser ” -> "Test eraser" OK

Python :
efficacite-4.py “ T e s t e r aser ” -> "Test eraser" OK
efficacite-134.py “ T e s t e r aser ” -> “T est eraser” NON OK
sobriete-102.py “ T e s t e r aser ” -> "Test eraser" OK

C :
sobriete-76.c “ T e s t e r aser ” -> "Test eraser" OK

Phrase en entrée : “ ”
Phrase attendue en sortie : “ ”

Java :
efficacite-131.java “ ” -> Erreur de compilation (Duplicate local variable i)
efficacite-75.java : “ ” -> “ ”
simplicite-28.java : “ ” -> “ ”
simplicite-54.java -> Erreur de compilation (Unreachable code)
simplicite-69.java : “ ” -> “ “
simplicite-82.java : “ ” -> “ ”
sobriete-92.java : “ ” -> “ ”

Python :
efficacite-4.py “ ” -> “ ”
efficacite-134.py “ ” -> “ ”
sobriete-102.py “ ” -> “ ”

C :
sobriete-76.c “ ” -> “ ”

Résumé

En résumé voici ce tableau récapitulant les algorithmes qui ne peuvent pas s'exécuter et ceux qui ne remplissent pas la tâche demandée :

	S'exécute / Compile	Enlève les espaces seuls
efficacite-131.java	X	X
efficacite-75.java	O	O
simplicite-28.java	O	O
simplicite-54.java	X	X
simplicite-69.java	O	X
simplicite-82.java	O	O
sobriete-92.java	O	O
efficacite-4.py	O	O
efficacite-134.py	O	X
sobriete-102.py	O	O
sobriete-76.c	O	O

O = Effectue l'action
X = N'effectue pas l'action

III - Complexité des algorithmes

efficacite-131.java

Cet algorithme ne compile pas, sa complexité est donc nulle (O(0)).

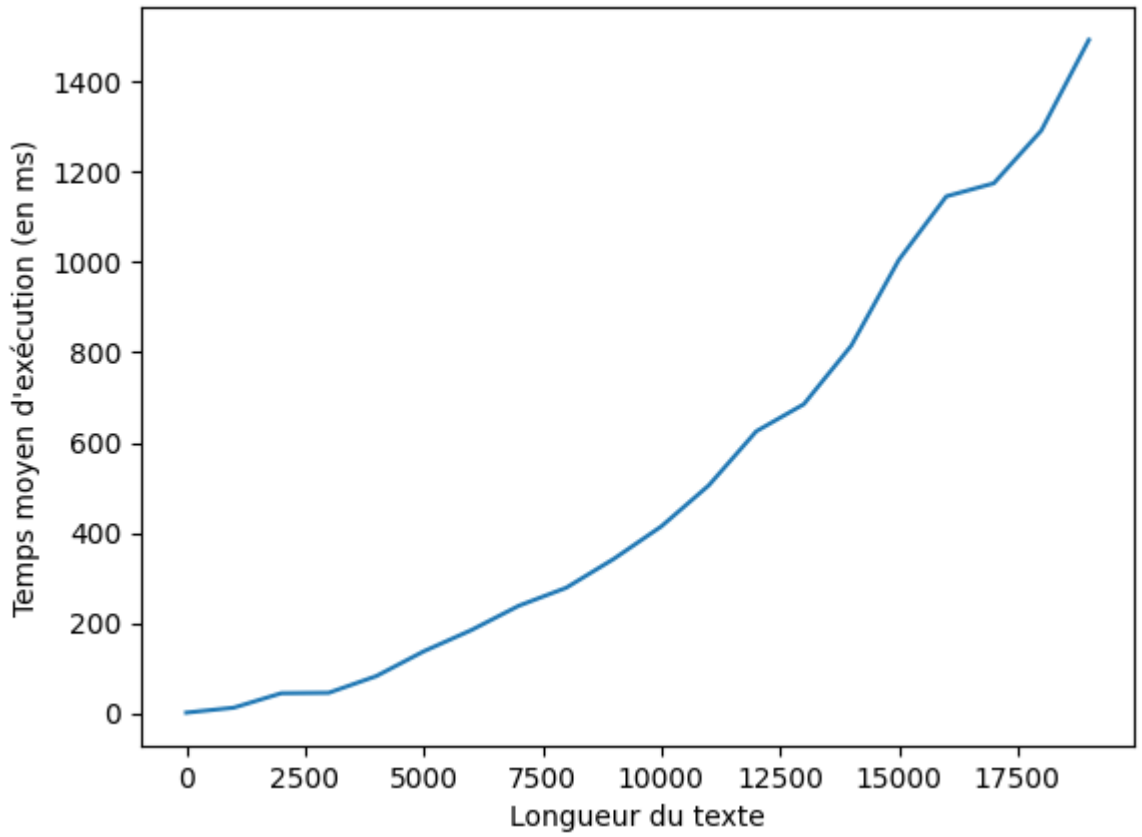
efficacite-75.java

Cet algorithme effectue une boucle for de la longueur du texte.
Dans ce for se trouve une boucle while, mais elle est compensée par le fait que tousles passages dans cette boucle while sont déduits des passages dans la boucle for (i+=compteur-1).
Cet algorithme va donc effectuer autant d'actions qu'il y a de caractères dans le texte de base. Dans le cas où le caractère n'est pas un espace, le programme va effectuer une seule action (ajouter à la nouvelle chaîne le caractère).
Dans le cas où le caractère est un espace, le programme va effectuer une affectation, (2 comparaisons, un ajout de caractère et une incrémentation) x le nombre d'espace + une comparaison et éventuellement un ajout de caractère et une incrémentation (dans le cas où il n'y a qu'un seul espace).
On considère que toutes les actions citées précedemment sont de complexité 1.
Le pire des scénarios serait celui du texte comportant un caractère et un espace "n n n n n n" par exemple.
Dans ce cas on effectuera 1+4+1+2 = 8 actions par espace, + 1 action par caractère, soit 9 actions pour 2 caractères.

La complexité de cet algorithme dans le pire des cas serait donc environ de O(9n/2), soit O(4,5n) avec n correspondant au nombre de caractères. Cela correspond à une complexité linéaire. Calculons maintenant le temps moyen d'exécution de l'algorithme :

```
Longueur 0 / Temps d'éxécution moyen : 0 millisecondes
Longueur 1000 / Temps d'éxécution moyen : 12 millisecondes
Longueur 2000 / Temps d'éxécution moyen : 44 millisecondes
Longueur 3000 / Temps d'éxécution moyen : 45 millisecondes
Longueur 4000 / Temps d'éxécution moyen : 82 millisecondes
Longueur 5000 / Temps d'éxécution moyen : 137 millisecondes
Longueur 6000 / Temps d'éxécution moyen : 184 millisecondes
Longueur 7000 / Temps d'éxécution moyen : 238 millisecondes
Longueur 8000 / Temps d'éxécution moyen : 278 millisecondes
Longueur 9000 / Temps d'éxécution moyen : 342 millisecondes
Longueur 10000 / Temps d'éxécution moyen : 414 millisecondes
Longueur 11000 / Temps d'éxécution moyen : 505 millisecondes
Longueur 12000 / Temps d'éxécution moyen : 625 millisecondes
Longueur 13000 / Temps d'éxécution moyen : 685 millisecondes
Longueur 14000 / Temps d'éxécution moyen : 815 millisecondes
Longueur 15000 / Temps d'éxécution moyen : 1005 millisecondes
Longueur 16000 / Temps d'éxécution moyen : 1146 millisecondes
Longueur 17000 / Temps d'éxécution moyen : 1175 millisecondes
Longueur 18000 / Temps d'éxécution moyen : 1292 millisecondes
Longueur 19000 / Temps d'éxécution moyen : 1493 millisecondes
```

Le ratio moyen est de 28 caractères par milliseconde, soit 28000 par seconde (dans l'intervalle [0,19000])
Si on trace la courbe représentative, on obtient cette courbe :



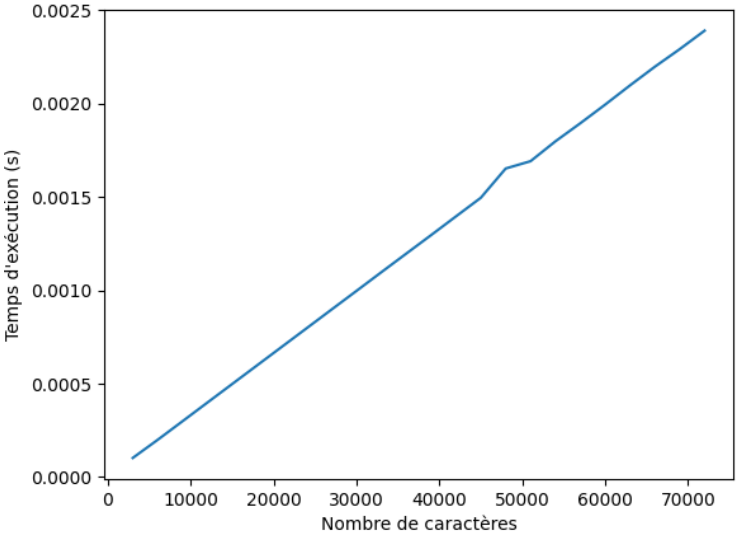
Nous pouvons observer une tendance exponentielle.
Ceci vient contredire notre théorie initiale, mais c'est normal. En effet, dans ce code, la nouvelle chaîne de caractère est construite avec la forme "str_erased += str.charAt(i)".
Ce += vient additionner la chaîne de caractère à elle-même, ce qui fait que plus elle grandit, plus le temps pour ajouter un caractère sera grand. Pour optimiser ce code, il faudrait utiliser une ArrayList au lieu d'une chaîne de caractère qu'on fait grandir.

efficacite-4.py

Pour cet algorithme, on parcourt la chaîne de caractère dans une boucle while, et un compteur est incrémenté à chaque espace trouvé. Si 1 espace est trouvé, on initialise j à 1, puis on rentre dans une boucle while qui va effectuer 2 comparaisons et 1 incrémentation, qui s'arrête quand elle trouve un caractère autre qu'un espace où qu'on arrive à la fin de la chaîne de caractère. A la fin de cette boucle while, une comparaison, un ajout de caractère et une incrémentation sont effectués. Le pire des scénarios serait celui du texte comportant un caractère et un espace "n n n n n n" par exemple.
On considère que toutes les actions citées précedemment sont de complexité 1.
Dans ce cas, on effectuera 1+3+3 = 7 actions par espace, + 1 action par caractère, soit 8 actions pour 2 caractères.
La complexité de cet algorithme dans le pire des cas serait donc environ de O(8n/2), soit O(4n) avec n correspondant au nombre de caractères.
Cela correspond à une complexité linéaire.

Calculons maintenant le temps moyen d'exécution de l'algorithme (avec la moyenne de 10 exécutions par longueur de texte) :

```
3000 : 0.0001026153564453125
6000 : 0.0001994609832763672
9000 : 0.000299072265625
12000 : 0.0003989219665527344
15000 : 0.0004986286163330078
18000 : 0.0005983591079711914
21000 : 0.0006981372833251953
24000 : 0.0007978439331054688
27000 : 0.0008977174758911133
30000 : 0.000997304916381836
33000 : 0.0010973453521728516
36000 : 0.001196742057800293
39000 : 0.0012959003448486327
42000 : 0.0013961315155029297
45000 : 0.0014959335327148437
48000 : 0.0016525983810424805
51000 : 0.001691722869873047
54000 : 0.0017981529235839844
57000 : 0.0018947839736938477
60000 : 0.0019946575164794924
63000 : 0.0020977020263671874
66000 : 0.002197670936584473
69000 : 0.0022917985916137695
72000 : 0.002390766143798828
```

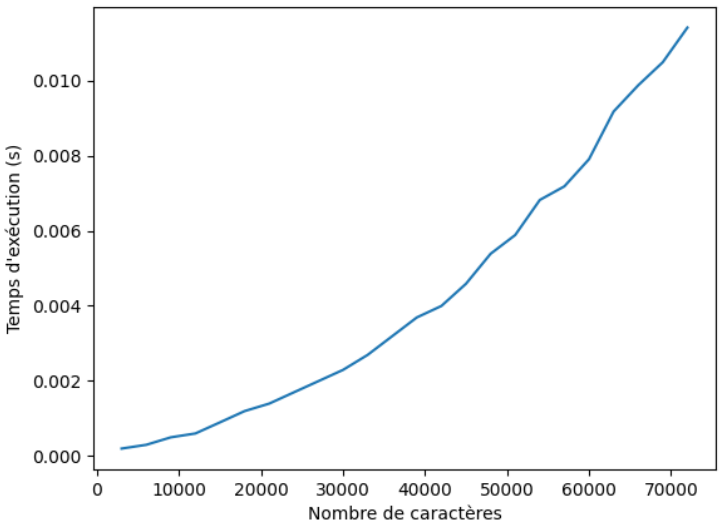


Nous pouvons donc observer une belle tendance linéaire de la part de cet algorithme, avec une moyenne de 30169122 caractères par seconde.

efficacite-134.py

J’avoue que je n’ai pas bien compris l’algorithme, d’autant plus qu’il ne marche pas pour tous les cas (voir II).
Pour avoir une idée sa complexité, nous allons calculer le temps d’exécution moyen

```
3000 : 0.0001994371414184570 3
6000 : 0.0002991914749145508
9000 : 0.000498652458190918
12000 : 0.0006015300750732422
15000 : 0.0008978605270385742
18000 : 0.001197075843811035
21000 : 0.0013964176177978516
24000 : 0.00169572830200195 3
27000 : 0.001994681358337402
30000 : 0.0022939205169677734
33000 : 0.002692842483520508
36000 : 0.003191542625427246
39000 : 0.003690171241760254
42000 : 0.003996062278747559
45000 : 0.004587507247924805
48000 : 0.005385565757751465
51000 : 0.005884289741516113
54000 : 0.006817960739135742
57000 : 0.00718083381652832
60000 : 0.007903313636779786
63000 : 0.00917224884033203
66000 : 0.009870600700378419
69000 : 0.01048588752746582
72000 : 0.011407709121704102
```



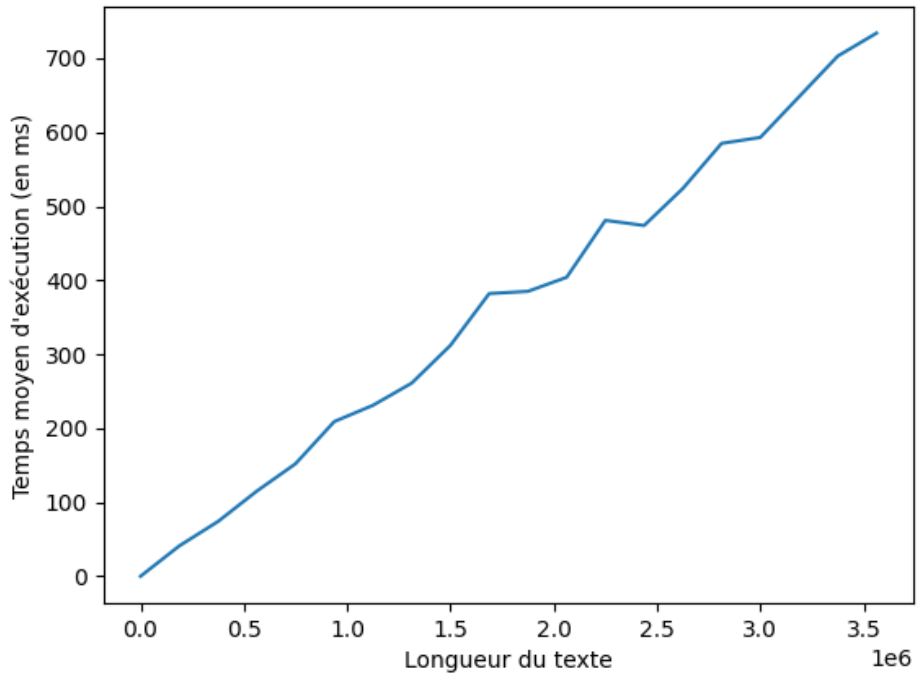
Nous observons donc une tendance exponentielle de cet algorithme, avec une moyenne de 11772590 caractères par seconde dans l’intervalle [0,72000]

sobriete-92.java

Cet algorithme est entièrement fait avec les regular expression (regex).
Pour calculer sa complexité, j'ai décidé de partir sur une méthode plus "pratique". J'ai créé une classe qui génère une chaîne de caractère aléatoire d'une longueur donnée en paramètre, et j'ai modifié la classe Main pour qu'il calcule le temps d'éxécution en millisecondes.
Pour chaque longueur de texte, j'ai fait la moyenne de 15 exécutions différentes pour avoir un résultat plus "fiable".
On obtient les résultats suivants :

```
Longueur 0 / Temps d'éxécution moyen : 0 millisecondes
Longueur 187500 / Temps d'éxécution moyen : 41 millisecondes
Longueur 375000 / Temps d'éxécution moyen : 74 millisecondes
Longueur 562500 / Temps d'éxécution moyen : 115 millisecondes
Longueur 750000 / Temps d'éxécution moyen : 152 millisecondes
Longueur 937500 / Temps d'éxécution moyen : 209 millisecondes
Longueur 1125000 / Temps d'éxécution moyen : 231 millisecondes
Longueur 1312500 / Temps d'éxécution moyen : 261 millisecondes
Longueur 1500000 / Temps d'éxécution moyen : 312 millisecondes
Longueur 1687500 / Temps d'éxécution moyen : 382 millisecondes
Longueur 1875000 / Temps d'éxécution moyen : 385 millisecondes
Longueur 2062500 / Temps d'éxécution moyen : 404 millisecondes
Longueur 2250000 / Temps d'éxécution moyen : 481 millisecondes
Longueur 2437500 / Temps d'éxécution moyen : 474 millisecondes
Longueur 2625000 / Temps d'éxécution moyen : 524 millisecondes
Longueur 2812500 / Temps d'éxécution moyen : 585 millisecondes
Longueur 3000000 / Temps d'éxécution moyen : 593 millisecondes
Longueur 3187500 / Temps d'éxécution moyen : 648 millisecondes
Longueur 3375000 / Temps d'éxécution moyen : 703 millisecondes
Longueur 3562500 / Temps d'éxécution moyen : 734 millisecondes
```

Si on trace la courbe représentative, on obtient cette courbe :

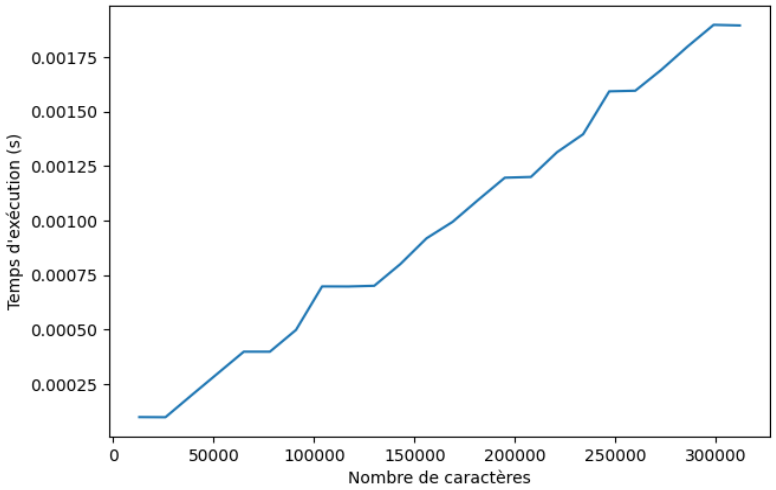


Nous pouvons donc constater une tendance linéaire de la complexité.
Le ratio moyen est de 4616 caractères par milliseconde, soit 4616000 par seconde.

sobriete-102.py

Cet algorithme est lui aussi composé de regular expression (regex). Pour calculer sa complexité, nous allons mesurer son temps d'exécution moyen et tracer sa courbe représentative :

```
13000 : 9.906291961669922e-05
26000 : 9.844303131103515e-05
39000 : 0.0001993894577026367
52000 : 0.0002991914749145508
65000 : 0.0003989696502685547
78000 : 0.0003988742828369140 6
91000 : 0.0004986763000488281
104000 : 0.0006983041763305664
117000 : 0.0006979703903198242
130000 : 0.0007012367248535156
143000 : 0.0008009433746337891
156000 : 0.0009188175201416016
169000 : 0.000994420051574707
182000 : 0.0010970354080200196
195000 : 0.0011968374252319335
208000 : 0.0012003183364868164
221000 : 0.0013136625289916991
234000 : 0.0013962745666503905
247000 : 0.0015928268432617188
260000 : 0.001595759391784668
273000 : 0.0016920804977416993
286000 : 0.0017982006072998046
299000 : 0.001897883415222168
312000 : 0.001894974708557129
```



Nous pouvons observer ici une tendance linéaire, avec une moyenne de 171851323 caractères par seconde.

Résumé

En résumé, nous avons 2 algorithmes de complexité exponentielle, 3 de complexité linéaire et 1 de complexité linéaire (car il ne compile pas).

Parmis les 3 de complexité linéaire, les plus efficaces sont :

efficacite-4.py -> 30169122 caractères en moyenne par seconde
sobriete-102.py -> 171851323 caractères en moyenne par seconde
sobriete-92.java -> 4616000 caractères en moyenne par seconde

Parmis les 2 de complexité exponentielle, les plus efficaces sont :

efficacite-134.py -> 27000 caractères en 0.001994s, soit 0.1994 ms
efficacite-75.java -> 1000 caractères en 0.012s, soit 12ms

IV - Simplicité des algorithmes

simplicite-54.java

Cet algorithme ne compilant pas, il sera dernier de la catégorie simplicité avec un score de 0.

simplicite-69.java

Cet algorithme n'est pas valide (voir II 2), il sera avant dernier de sa catégorie avec un score de 1, juste devant l'algorithme ne compilant pas.

simplicite-28.java

Cet algorithme n'a aucun commentaire, il sera donc compliqué à comprendre par un nouveau programmeur qui reprend le code. De plus, le code utilise UTF-8 pour comparer des chaînes de caractère (str.charAt(i) == 32), ce qui peut être déroutant pour un jeune programmeur qui n'a pas l'habitude de cette notation. Enfin, les boucles if else imbriquées rendent le code difficile à comprendre, d'autant plus qu'il n'y a aucun commentaire pour comprendre l'utilité de ces if et else.

simplicite-82.java

Cet algorithme contient des commentaires, est plutôt facile à lire et en plus est interchangeable ! Par exemple, si nous souhaitons enlever 2 espaces au lieu de 1 espace seulement il suffit juste de changer une seule variable. Ceci est un gros point fort de ce programme, car si quelqu'un veut reprendre le code pour effectuer cette modification il aura juste à modifier une valeur, et sans relire tout le code car il est bien commenté. Pour les arguments cités précedemment je mettrai cet algorithme en premier dans mon classement des algorithmes simples.

V - Tableau d’évaluation final

Evaluation	Algo	Lisibilité	Qualité	Temps d’éxécution	Complexité	Sobriété
5	simplicite-82.java	++++	+++	N/A	N/A	N/A
3	simplicite-28.java	+	++	N/A	N/A	N/A
1	simplicite-69.java	-	- -	N/A	N/A	N/A
0	simplicite-54.java	N/A	N/A	- - - -	- - - -	N/A
5	efficacite-4.py	N/A	N/A	+++++	+++++	N/A
4	sobriete-76.c	N/A	N/A	+++	++++	++
4	sobriete-102.py	N/A	N/A	++++	+++	+++++
3	sobriete-92.java	N/A	N/A	++	++	N/A
2	efficacite-75.java	N/A	N/A	-	-	N/A
1	efficacite-134.py	N/A	N/A	- -	- -	N/A
0	efficacite-131.java	N/A	N/A	- - - -	- - - -	N/A

Le premier dans la catégorie simplicité est simplicite-82.java
Le premier dans la catégorie efficacité est efficacite-4.py