TP 1 - Profilage.

Remarque : les nombres suggérés pour les tests dans les exercices suivants sont donnés à titre indicatif. N'hésitez pas à les adapter à votre machine en fonction de ses capacités.

Exercice 1 (`timeit`)

Le but de cet exercice est d'identifier automatiquement les algorithmes les plus lents d'un ensemble de modules, dans une situation où l'on ne dispose pas d'un programme complet à exécuter. Pour ce faire, commencez par récupérer :

1. l'archive `algorithmes-de-tri-2023-05-22.zip`, qui contient un grand nombre d'algorithmes de tri répartis en modules [1] ;
2. le fichier `benchmark_tris_init.py`, qui récupère automatiquement la plupart des fonctions de tri de ces modules, en se limitant notamment aux fonctions dont le seul paramètre est l'itérable à trier.

(a) Complétez le fichier `benchmark_tris_init.py` de manière à pouvoir générer un classement des algorithmes proposés, triés par ordre croissant de temps d'exécution. Utilisez le module `timeit` et écrivez une fonction générant une liste aléatoire de n entiers pour vos tests. Le résultat ressemblera à ceci :

```
$ # dans le répertoire contenant les algorithmes de tri
$ python3 benchmark_tris_correction.py
Évaluation de toutes les fonctions de tri sur 1000 instances
aléatoires de taille 10
(moyennes prises sur 10 essais)
Traité 36 / 36 modules.

Résultats par temps d'exécution moyen croissant (secondes):

      genial_sort : 5.767447757534683e-06
moins_genial_sort : 7.141655194573105e-06
              ...
         nul_sort : 12345.6789
```

(b) Fixez un petit nombre d'instances (par exemple : 10) et augmentez progressivement la taille des instances à générer (par exemple : 100, 200, 300, ...). Que constatez-vous ?

(c) Fixez maintenant la taille des listes (par exemple : 10) et augmentez progressivement le nombre d'instances à générer (par exemple : 100, 1 000, 10 000, 20 000, ...). Quels sont les cinq pires algorithmes ?

Exercice 2 (`cProfile`)

Récupérez l'archive `compression-2023-05-22.zip` sur la page du cours. Cette archive provient de la même source que les algorithmes de tri testés plus haut, et contient plusieurs algorithmes de compression de données. Nous allons maintenant utiliser `cProfile` pour améliorer la fonction `run_length_encode` du module `run_length_encoding`, qui renvoie l'encodage RLE d'un texte donné.

(a) Écrivez un programme `benchmark_run_length_encoding.py` appelant la fonction `run_length_encode` sur une chaîne aléatoire. Choisissez une taille de chaîne suffisamment grande pour que le programme soit "lent" (disons, environ 10 secondes d'exécution sur votre machine selon la commande `time`).

1. Source : `https://github.com/TheAlgorithms/Python`, version du 22/05/2023.

(b) Utilisez `cProfile` pour déterminer les fonctions qui sont le plus appelées dans le code. Attention, ceci peut fortement ralentir l'exécution de votre programme : veillez à réduire la taille de la chaîne générée par rapport à la sous-question précédente. Quelle(s) optimisation(s) pouvez-vous en déduire ?

(c) Appliquez votre modification et enregistrez-la dans un fichier différent. Modifiez votre programme pour qu'il compare les temps d'exécution des deux versions avec `timeit` sur la même chaine aléatoire. Par exemple :

```
$ python3 benchmark_run_length_encoding.py
run_length_encoding.run_length_encode:   11.817009345046245
run_length_encoding_2.run_length_encode: 11.083991175983101
```

Exercice 3 (`line_profiler`)

Reprenez votre dernier fichier optimisé de l'exercice précédent.

(a) Utilisez `line_profiler` pour repérer les endroits de `run_length_encode` les plus gourmands en temps d'exécution. Ceci ralentit considérablement le code, exécutez donc vos tests sur une chaîne beaucoup plus petite (par exemple de taille 1 000).

(b) Sur base du rapport de `line_profiler`, proposez une optimisation. Implémentez-la dans un nouveau fichier, et comparez le résultat aux versions précédentes en modifiant à nouveau votre programme `benchmark_run_length_encoding.py`.

Exercice 4 (`memray`)

Nous allons maintenant évaluer les besoins en mémoire de l'implémentation de l'algorithme de compression LZ77 fournie dans le fichier `lz77.py`.

(a) Écrivez un programme `benchmark_lz77.py` qui lance l'algorithme sur une chaîne aléatoire suffisamment grande, et analysez sa consommation avec `memray`.

(b) Utilisez `memray` pour déterminer :

1. les 5 endroits responsables des plus grosses allocations en mémoire ;
2. les 5 endroits responsables des plus nombreuses allocations en mémoire.

(c) **Sans les implémenter** [2], proposez des pistes d'amélioration pour réduire la consommation en mémoire ou le nombre d'allocations aux endroits repérés à la sous-question précédente.

2. Si vous ne pouvez pas vous en empêcher : attention à l'impact éventuel sur les performances !