Comparaison de la complexité de deux types Python (`list` et `dict`)

L'objectif de cette séance de comparer les types `list` et `dict` du langage Python, en termes de complexité notamment pour la recherche d'un élément. Nous allons stocker dans ces structures les notes associées à des étudiants, et il s'agira de retrouver le plus rapidement possible la note d'un étudiant en donnant son nom (codé sur 8 caractères).

Nous allons éditer un fichier `compare.py`, qui utilisera le module `time_profiler.py` déjà fourni.

1 Les types Python à comparer

1.1 Les listes : `list`

Nous avons vu que la fonction `append()` était particulièrement efficace.
Pour mettre 20 à l'étudiant "superpro", il suffit de faire :

```
ln = list()  # list of names
lm = list()  # list of marks

ln.append("superpro")
lm.append(20)
```

Il faut donc 2 listes, avec les éléments insérés dans le même ordre...

Alors nous pouvons faire :
- `ln.index(''superpro'')` pour obtenir l'indice correspondant à l'élément cherché ;
- `lm[i]` pour accéder l'élément d'indice `i` de la liste `lm`.

Question 1 : (TD et TP) – Écrivez une fonction `search_list(name)` qui retourne la note de l'étudiant nommé `name` (cela peut se faire en une ligne...).

1.2 Les dictionnaires : `dict`

L'utilisation d'un dictionnaire est encore plus simple :

```
d = dict()

d["superpro"] = 20
```

permet d'associer la note 20 à l'étudiant "superpro".

Et pour retrouver la note de l'étudiant nommé `name`, cela se fait en une ligne :

```
def search_dict(name):
    return d[name]
```

Question 2 : (TD et TP) – Écrivez une fonction `insert_both(name, mark)` qui rajoute l'étudiant `name` avec sa note `mark` à la fois dans la liste et le dictionnaire.

2 Construction des structures

Avant de rechercher un élément particulier, il nous faut remplir les structures (liste et dictionnaire) avec un grand nombre d'éléments. Le mieux pour cela est de faire un tirage aléatoire du nom et de la note. Voici l'extrait de programme correspondant :

```
import random
import string

def get_random_name(length):
    return ''.join(random.choice(string.ascii_uppercase) for i in range(length))

def get_random_mark():
    return random.randint(0, 20)

for n in range(10000):
    name = get_random_name(8)
    mark = get_random_mark()
    insert_both(name, mark)
```

Le début est particulièrement compact, car il utilise une forme fonctionnelle.
Le code "classique" équivalent serait :

```
  def get_random_name(length):
      result = ''
      letters = string.ascii_uppercase
      for i in range(length):
          l = random.choice(letters)
          result = result + l
      return result
```

3 Profil temporel

L'extrait de programme suivant permet de rechercher, pour chaque étudiant dont le nom est dans la liste `lm`, sa note dans la liste et dans le dictionnaire, en s'assurant qu'on a bien la même valeur. Et on fait le profil temporel de tout cela.

```
  import time_profiler

  time_profiler.start()

  for name in ln:
      note1 = search_list(name)
      note2 = search_dict(name)
      assert(note1 == note2)

  time_profiler.stop()
```

Question 3 : (TD et TP) – Complétez le programme `compare.py`.

Question 4 : (TP) – Lancez ce programme et analysez son profil temporel. Quelle structure de données est la plus rapide, du point de vue de la recherche d'un élément ?

4 Complexité

Question 5 : (TD et TP) – Modifiez le programme pour qu'il se comporte ainsi :

1. création de structures avec 10000 éléments aléatoires ;
2. profilage temporel de la recherche de chacun des éléments ;
3. rajout de 10000 nouveaux éléments aléatoires dans les structures ;
4. nouveau profilage temporel.

Question 6 : (TP) – Observez que le temps de `search_list()` a quadruplé, tandis que le temps de `search_dict()` a doublé. En déduire la complexité de chacune de ces fonctions (attention, il y a un piège : la complexité de la fonction de recherche d'un élément n'est pas celle de la fonction de recherche de tous les éléments – N fois plus complexe !).

Question 7 : (TP, pour les plus rapides) – Vérifiez que l'insertion d'un nouvel élément se fait en temps constant.