1 Recherche dans une liste : problème d'efficacité

Les listes, en tant que structures de données séquentielles, sont pratiques pour parcourir les éléments de la liste (ex : `for element in liste: ...`) mais ne sont pas optimisées pour la recherche d'un élément dans la liste.
En effet **si la liste n'est pas triée, la recherche d'un élément particulier a un coût linéaire puisqu'il peut être nécessaire de parcourir l'intégralité de la liste** si l'élément ne s'y trouve pas !

Rappel d'un algo de recherche séquentielle en Python :

1. Version parcours par indice :

```
def recherche(tab, valeur):
    for i in range(len(tab)):
        if tab[i] == valeur:
            return i # position de l'élément recherché dans le tableau
    return -1 # recherche infructueuse
```

2. Version parcours par élément :

```
def recherche(tab, valeur):
    for element in tab:
        if element == valeur:
            return True # l'élément recherché est présent dans le tableau
    return False # l'élément recherché est absent du tableau
```

De façon différente, les dictionnaires sont une structure de données optimisée pour la recherche efficace d'un élément dans une collection.

> La recherche d'un élément dans un dictionnaire s'effectue en effet avec un coût en temps d'exécution constant.

L'accès à un élément est aussi rapide.

2 Dictionnaires en Python

Les dictionnaires (déjà étudiés en classe de 1ère), aussi appelés **tableau associatif** ou **p-uplets nommés**, permettent de **stocker des valeurs** et d'**y accéder au moyen d'une clé**, contrairement aux listes Python qui permettent d'accéder à une donnée au moyen d'un indice.
Exemple :

```
dico = dict()
dico["Nom"] = "Jahier"
dico["Prenom"] = "Erwan"
print(f"Bonjour {dico["Prenom"]} {dico["Nom"]}, comment vas-tu ?")
```

Dans cet exemple, les **clés** sont les chaînes `"Nom"` et `"Prenom"`.
On exploite ensuite leurs **valeurs**, dans une chaîne de caractères formatée.

Les opérations classiques que l'on peut effectuer sur un dictionnaire sont :
- Ajouter une nouvelle entrée au dictionnaire en créant une nouvelle clé.
- Modifier la valeur associée à une clé existante.
- Supprimer une entrée dans un dictionnaire.
- Rechercher la présence d'une clé dans un dictionnaire.

La recherche dans un dictionnaire est optimisée pour s'effectuer sur les clés et non sur les valeurs. Par exemple avec le dictionnaire que nous avons créé précédemment, la commande `"Nom" in dico` vaut True alors que `"Jahier" in dico` vaut False.

Dans un dictionnaire, les clés et les valeurs ne jouent donc pas du tout le même rôle et ne sont pas interchangeables.

Une clé peut être d'un autre type que chaîne de caractères, du moment que c'est un **objet non mutable**, c'est à dire qui ne peut pas être modifié. Une clé ne peut pas être une liste par exemple car une liste est un objet mutable que l'on peut modifier, par exemple au travers de la méthode .append().

Regardons ce qui se passe si on essaye de définir une clé de type list pour un dictionnaire :

```
>>> dico[[2,1]] = "toto"
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-d463baccae6e> in <module>()
----> 1 dico[[2,1]]

TypeError: unhashable type: 'list'
```

Le type list n'est pas « hashable ». Mais qu'est-ce que le hachage ?

3 Hachage

La notion de hachage est omniprésente en informatique et est au cœur du fonctionnement des dictionnaires. **Le hachage est un mécanisme permettant de transformer la clé en un nombre unique permettant l'accès à la donnée, un peu à la manière d'un indice dans un tableau.**

Une **fonction de hachage** est une fonction qui **renvoie une valeur unique** (empreinte) **à partir de la donnée** fournie en entrée.

Elle doit respecter les règles suivantes :
- Des données identiques doivent donner des empreintes identiques.
- Des données différentes doivent donner dans la mesure du possible des empreintes différentes.
- Connaissant l'empreinte, il ne doit pas être possible de reconstituer la donnée d'origine.
- La longueur de l'empreinte (valeur retournée par la fonction de hachage) doit être toujours la même, indépendamment de la donnée fournie en entrée.

Exemples d'utilisations du hachage :
- stockage des mots de passe dans un système informatique un peu sécurisé. En effet, lorsqu'on crée un compte sur un service en ligne, le mot de passe ne doit pas être stocké en clair, une empreinte est générée afin que si le service est piraté et que les comptes sont dérobés, il ne soit pas possible de reconstituer le mot de passe à partir de l'empreinte.
- détection de la modification d'un fichier (vous avez peut-être déjà vérifié la somme md5 après le téléchargement d'une fichier pour contrôler son intégrité).

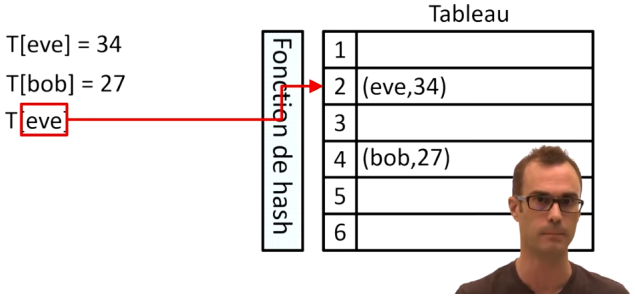
Remarque : en Python, vous pouvez avoir accès à une fonction de hachage facilement grâce à la fonction hash() :

```
>>> hash("Jahier")
3736790693017685111
>>> hash("jahier")
1049854859100975814
```

Vérifiez à l'aide de quelques exemples que cette fonction hash() est bien une fonction de hachage.

Regardez la vidéo ci-dessous sur les tables de hachage :

`https://www.youtube.com/watch?v=IhJo8sXLfVw&t=1s`



Ce qui est important à retenir c'est que la recherche dans une table de hachage est indépendante du nombre d'éléments dans cette table. Dans une liste chaînée au contraire, la recherche d'un élément prend un temps proportionnel au nombre d'éléments.

> Dans un dictionnaire, les clés sont stockées dans une table de hachage, ce qui explique le fait que le dictionnaire est optimisé pour la recherche sur les clés.

Complément : rappel de cours sur l'utlisation des dictionnaires en Python :
`https://www.youtube.com/watch?v=VnhBoQAgIVs`