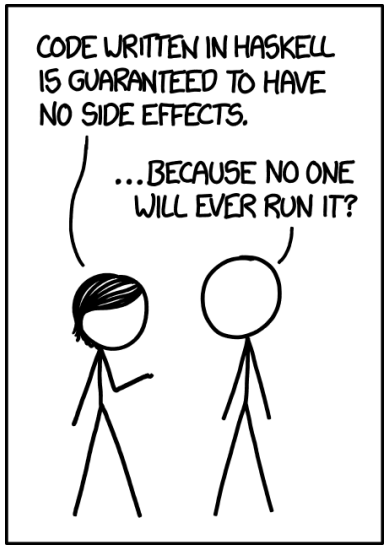
TP Info : Listes et effets de bords



1 Rappels sur les listes, effets de bord

Dans ce qui suit, `t1` et `t2` désignent deux listes et `a` désigne un nombre. Dans les exemples ci-dessous, on supposera que $t_1 = [1, 2, 7]$, $t_2 = [1, 9]$ et $a = 3$.

- `t1[0] = a` remplace le premier élément de la liste t_1 par a. On dit que la liste t_1 a été *mutée* et que cette opération agit par *effet de bord*. Dans notre exemple, une telle instruction tranforme la liste t_1 en $[3, 2, 7]$.
- `t1.append(a)` ajoute la valeur a en queue de liste. Cette méthode agit par effet de bord. Dans notre cas, cette instruction modifie t_1 en $[1, 2, 7, 3]$.
- `t1 + t2` *crée* une nouvelle liste obtenue par concatenation des listes `t1` et `t2`. Dans notre cas, `t1 + t2` crée la liste $[1, 2, 7, 1, 9]$. Les listes t_1 et t_2 sont inchangées après l'execution de cette instruction.

Plus généralement, on dit qu'une fonction `foo` agit par effet de bord sur son argument `t` si la valeur de `t` est modifiée par l'appel de `foo(t)`. Par exemple, la suite d'instructions

```
def foo(t):
    t[0] = 3

t = [1, 2]
foo(t)
print(t)
```

affiche `[3, 2]`. Cela prouve que la valeur de `t` a été modifiée : `foo` a eu un effet de bord sur son argument `t`. Une fonction qui ne provoque pas d'effet de bord est dite pure.

Enfin, nous utiliserons le fait que la boucle `for x in t1` fait parcourir à `x` les éléments de la liste `t1`. Dans notre exemple, la boucle donne à `x` successivement les valeurs 1, 2 et 7.

1. Tester chacune des fonctions suivantes sur quelques exemples de votre choix et dire si elles ont un effet de bord sur leur argument. Bien entendu, `x` désigne un entier, tandis que `t` désigne une liste.

```
def foo_0(x):
    x += 1
    return x
```

```
def foo_1(t):
    t[0] = t[0] + 1
```

```
def foo_1_bis(x):
    x = x + 1
```

```
def foo_2(t):
    t.append(0)
```

```
def foo_2_bis(t):
    t = t + [0]
```

```
def foo_2_doublebis(t):
    t += [0]
```

```
def foo_3(t):
    t_local = t
    t_local[0] = t_local[0] + 1
```

```
def foo_3_bis(t):
    t_local = [x for x in t]
    t_local[0] = t_local[0] + 1
```

2. (a) Écrire une fonction `stop_zero_1(t, eps)` qui renvoie, sans modifier t, une liste contenant les mêmes éléments que la liste t, mais où les éventuelles cases contenant 0 sont remplacées par une case contenant `eps`.

   (b) Écrire une fonction `stop_zero_2(t, eps)` qui ne renvoie rien, mais modifie la liste L, en remplaçant tous les zéros éventuels par `eps`.

3. *Moyenne-écart-type.*

   (a) Écrire une fonction `moyenne(t)` qui renvoie la moyenne d'une liste de nombres t.

   (b) On rappelle que si $t = [t_0, \dots, t_{n-1}]$ est une liste de nombres de moyenne égale à $\bar{x}$, alors la variance de t est la moyenne de la liste

$$t = \left[(t_0 - \bar{x})^2, \dots, (t_{n-1} - \bar{x})^2\right].$$

   L'écart-type est la racine carrée de la variance. Écrire une fonction `ecart_type(t)` qui renvoie l'écart-type d'une liste de nombres `t`.

2 Algorithme de compression RLE

On cherche à mettre en place un algorithme permettant de coder une liste $l = [l_0, \dots, l_{n-1}]$ sous une forme qui prendra en général moins de place en mémoire. Le principe est de repérer les sous-listes constantes et de coder chaque sous-liste constante par la valeur de la constante et la la taille de la sous-liste. C'est-à-dire que la liste :

$$l = [\underbrace{v_0, \dots, v_0}_{t_0 \text{ fois}}, \underbrace{v_1, \dots, v_1}_{t_1 \text{ fois}}, \dots, \underbrace{v_{p-1}, \dots, v_{p-1}}_{t_{p-1} \text{ fois}}]$$

sera codée par la liste de couples :

$$c = [(v_0, t_0), (v_1, t_1), \dots, (v_{p-1}, t_{p-1})]\,.$$

1. *Exemples :* Quel est le codage RLE de la liste $[4, 4, 1, 1, 1, 1, 1, 3, 3, 3, 3, 6, 6, 6, 6]$? Quel est celui de la liste $[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]$? Quelle est la liste dont le codage RLE est $[(3, 5), (4, 2), (1, 3), (3, 1)]$?
2. *Décodage :* Écrire une fonction `decode_rle(c)` qui reconstitue et renvoie la liste l de départ à partir de la liste codée c. Vérifier la validité de cette fonction sur les exemples ci-dessus.
3. *Codage :* Écrire une fonction `encode_rle(l)` qui renvoie la liste c correspondant au codage RLE de la liste l donnée en en argument. Vérifier la validité de cette fonction sur les exemples ci-dessus.
4. Discuter de l'intérêt de ce codage suivant la forme de la liste à coder.

3 Manipulations de parties

Dans cette partie, on considère un ensemble fini $E = \{a_0, \dots, a_{n-1}\}$ où les a_k sont deux à deux distincts. On peut représenter une partie A de E par une liste d'indices $k \in [\![0, n-1]\!]$ tels que $a_k \in A$. Par exemple, si $E = \{a_0, a_1, \dots, a_{12}\}$, alors $A = \{a_1, a_9, a_2, a_7, a_8\}$ peut être représenté par la liste $[1, 9, 2, 1, 7, 8]$ mais aussi par la liste $[1, 2, 7, 8, 9]$.

3.1 Représentation par des listes.

1. Écrire une fonction `sans_doublons(t)` qui prend comme argument une liste d'entiers et renvoie une liste qui contient une fois et une seule chacun des éléments de `t` sans aucune répétition.

La méthode `.sort()` permet de trier une liste. Dans la suite, on supposera donc que les listes représentant une partie A de E sont triées et sans doublons.

2. (a) Écrire une fonction `est_present(t, x)` renvoyant le booléen `True` si x et un élément de t et renvoyant `False` sinon. Cette fonction implémentera l'algorithme de recherche dichotomique.
   (b) Montrer que si n est la longueur de la liste t et l_k est le nombre d'éléments dans l'intervalle de recherche après k itérations, alors
   $$l_k \leqslant \frac{n}{2^k}$$
   (c) En déduire qu'il existe $c_1, c_2 \geqslant 0$ tels que le nombre d'accès mémoire dans le tableau t lors de l'exécution de `est_present(t, x)` est majoré par $c_1 \ln(n) + c_2$.
3. On suppose maintenant que les listes t_A et t_B sont triées par ordre croissant. Proposer une fonction `intersection` qui prend en arguments deux listes t_A et t_B et renvoie l'intersection de ces deux listes. On veillera à ce qu'elle n'utilise pas plus de `len(ta) + len(tb)` accès mémoire.

3.2 Représentation binaire

On peut également représenter une partie de l'ensemble $E = \{a_0, \dots, a_{n-1}\}$ par une liste de 0 ou de 1. Une partie A sera alors représentée par la liste d'entiers t_A de longueur n définie par $t_A[i] = 1$ si $a_i \in A$, et $t_A[i] = 0$ si $a_i \notin A$. Par exemple, si $E = \{a_0, \dots, a_5\}$ alors la liste $t_A = [0, 0, 1, 0, 1, 1]$ représente la partie $\{a_2, a_4, a_5\}$.

1. Écrire la fonction `barre` qui prend en argument une liste de bits représentant une partie A et renvoie la liste de bits représentant le complémentaire de A.
2. Écrire les fonctions `inter`, `union`, `delta` qui permettent de calculer respectivement les listes de bits représentant la réunion, l'intersection, la différence symétrique de deux parties représentées par leurs listes de bits.
3. Vérifier que ces fonctions sont de complexité linéaire. C'est-à-dire que le nombre de lectures de cases effectuées par ces fonctions est majoré par un multiple de la longueur des listes.

3.3 Représentation par un entier

Grâce à la représentation précédente, on peut représenter une partie de E par un nombre entier compris entre 0 et $2^n - 1$. Si la partie A est représentée par la liste de bits $t_A = [b_0, ..., b_{n-1}]$, alors A peut aussi être représentée par l'entier p_A qui s'écrit en base 2 :

$$p_A = \underline{b_{n-1} \dots b_0}_{\,2}.$$

1. On suppose que $E = \{a_0, \dots, a_7\}$. Quelles sont les parties correspondant aux entiers 63 et 124 ?

On suppose dans toute la suite que l'ensemble E est représenté par la liste `e` de ses éléments, et que son cardinal est stocké dans la variable `n`. On utilisera `E` et `n` comme des variables globales.

2. Écrire une fonction `entier_vers_liste` qui prend comme un argument un entier représentant une partie A de E et renvoie la liste des éléments de A.
3. On suppose que `pa` et `pb` sont deux entiers représentant respectivement une partie A et une partie B de E. Écrire la fonction `inter` de sorte que `inter(pa, pb)` renvoie le nombre entier représentant la partie $A \cap B$.
4. *Énumération des parties :*
   (a) Écrire une fonction `plus_un(t, n)` qui prend comme argument la liste t des n bits de l'écriture en base 2 d'un entier p compris entre 0 et $2^n - 2$ et renvoie la liste des bits représentant l'entier $p + 1$.
   (b) Écrire une fonction `bits_parties(n)` qui affiche (dans l'ordre des bits de poids faible) les n bits de l'écriture en base 2 des nombres $0, 1, \dots, 2^n - 1$.
   (c) S'inspirer de la fonction précédente pour écrire une fonction `parties` qui affiche successivement toutes les parties de E. (On affichera leur liste d'éléments).