ÉNUMÉRATIONS

TP

Les exercices de cette feuille consistent à énumérer des familles de listes ou d'uplets. Certaines applications sont proposées immédiatement, mais la plupart ne seront vues que plus tard.

Partie I — Couples, triplets, etc.

Exercice 1

1) Écrire un programme `CouplesDansUne(L)` qui fabrique la liste des couples d'éléments de `L`.

2) Écrire un programme `CouplesDansDeux(A, B)` qui fabrique la liste de tous les couples `(a, b)` avec a dans `A` et b dans `B`.

3) Synthétiser tout ceci en un seul programme `Couples(A, B = None)` qui fait comme le précédent, mais si l'argument `B` n'est pas précisé fait comme dans la première question.

Exercice 2 — Écrire de même les programmes `TripletsDansUne(L)` et `TripletsDansTrois(A, B, C)`.

Exercice 3

1) Combien y a-t-il de nulluplets à composantes dans une liste de longueur n ?

2) En remarquant que les $(k+1)$-uplets d'un ensemble s'obtiennent à partir des k-uplets, écrire le programme `UpletsDansUne(L, k)`.

3) De même écrire le programme `UpletsDansPlusieurs(Listes)` qui prend en argument une liste de k listes, et renvoie la liste des k-uplets de la forme $(x_0, x_1, \ldots, x_{k-1})$ avec x_i dans `Listes[i]`.

Exercice 4 — Écrire un programme qui renvoie tous les triplets $(a, b, c) \in (\mathbf{N}^*)^3$ tels que $a^2 + b^2 = c^2$ et dont les trois composantes n'excèdent pas une constante M donnée.

Partie II — Parties

Exercice 5 — Écrire le programme `Parties(L)`. Il n'est obligé de renvoyer le résultat dans l'ordre ci-dessous, et on suppose que tous les éléments de `L` sont distincts.

```
>>> Parties([1, 2, 3])
[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
```

Exercice 6 — Odomètre. Appelons *odomètre* une liste à valeurs dans $\{0\,;\,1\}$. Un odomètre peut être incrémenté : on augmente de 1 la case d'indice zéro. Si elle arrive à 2, on la remet à zéro et on augmente la case d'indice un. Si elle-même arrive à 2, on la remet à zéro et on augmente la case d'indice deux, etc.. Lorsqu'on essaie d'incrémenter un odomètre avec « que des uns », on convient qu'il se coince à la position ci-dessous.

```
>>> K = [0, 0, 0, 0, 0] ; IncrémenterOdomètre(K) ; K
[1, 0, 0, 0, 0]
>>> K = [1, 1, 1, 0, 1] ; IncrémenterOdomètre(K) ; K
```

```
[0, 0, 0, 1, 1]
>>> K = [1, 1, 1, 1, 1] ; IncrémenterOdomètre(K) ; K
[0, 0, 0, 0, 2]
```

Écrire le programme `IncrémenterOdomètre(K)`. Il modifie la liste `K` et ne renvoie rien.

Exercice 7

1) Soit `L` une liste et `K` un odomètre de même taille. Écrire un programme `Partie(L, K)` qui fabrique la sous-liste de `L` obtenue en ne gardant que les termes d'indices i pour lesquels `K[`i`]` $= 1$.

```
>>> Partie([0, 1, 2, 3, 4, 5], [1, 1, 0, 0, 1, 0])
[0, 1, 4]
```

2) En déduire une nouvelle version du programme `Parties(L)`.

Exercice 8 — Déduire de ce qui précède un programme `EstSomme(L,` s`)` qui détermine s'il est possible d'écrire s comme une somme d'élément de `L`, chaque élément ne pouvant servir qu'une seule fois (sauf si bien sûr il apparaît en plusieurs exemplaires dans `L`).

Exercice 9 — Odomètre trafiqué. Écrire un programme `IncrémenterÀUnsConstants(K)` qui bascule directement un odomètre à la prochaine valeur affichant autant de 1 que celle de départ. À nouveau, ce programme ne renvoie rien et modifie la liste `K`.

```
>>> K = [1, 1, 1, 0, 0] ; IncrémenterÀUnsConstants(K) ; K
[1, 1, 0, 1, 0]
>>> K = [0, 1, 1, 1, 0] ; IncrémenterÀUnsConstants(K) ; K
[1, 1, 0, 0, 1]
>>> K = [0, 0, 1, 1, 1] ; IncrémenterÀUnsConstants(K) ; K
[0, 0, 0, 0, 2]
```

Exercice 10 — Déduire de ce qui précède le programme `PartiesDeCardinalDonné(L,` k`)` qui fait ce qu'on imagine.

Partie III — Anagrammes, permutations et arrangements

Exercice 11 — Écrire un programme `Distincts(L)` qui renvoie la liste des éléments de `L`, en supprimant les répétitions.

Exercice 12 — Anagrammes. On souhaite ici produire la liste de toutes les anagrammes d'une liste donnée, sans répétition.

```
>>> Anagrammes([1, 1, 2, 3])
[[1, 1, 2, 3], [1, 1, 3, 2], [1, 2, 1, 3], [1, 3, 1, 2],
 [1, 2, 3, 1], [1, 3, 2, 1], [2, 1, 1, 3], [3, 1, 1, 2],
 [2, 1, 3, 1], [3, 1, 2, 1], [2, 3, 1, 1], [3, 2, 1, 1]]
```

1) Écrire un programme `CasesVides(`a`)` qui étant donnée une liste a renvoie la liste des indices i tels que a`[`i`] == None`.

2) Appelons *anagramme partielle* une liste dont certaines cases sont vides. On se donne une anagramme partielle a, un objet x et un entier k inférieur ou égal au nombre de cases vides de a. Écrire le programme `AjouterÀUne(`a`,` x`,` k`)` qui renvoie la liste de toutes les anagrammes partielles obtenues en ajoutant k exemplaires de x à a.

3) En déduire un programme `AjouterÀToutes(A,` x`,` k`)` qui étant donnée une liste `A` d'anagrammes partielles renvoie la liste de toutes les anagrammes partielles obtenues en ajoutant k exemplaires de x à un élément de `A`.

4) Interlude : écrire le programme `Occurrences(L, x)` qui calcule et renvoie le nombre d'occurrences de x dans `L`.

5) Écrire finalement le programme `Anagrammes(L)`.

Exercice 13

1) Tester les commandes ci-dessous.

```
>>> L = list("HURLUBERLU") ; L
>>> s = "".join(L) ; s
```

2) En déduire le programme `AnagrammesEnChaîne(s)` qui construit la liste de toutes les anagrammes de la chaîne s.

```
>>> AnagrammesEnChaîne("KIWI")
["KIIW", "KIWI", "KWII", "IKIW", "IKWI", "IWIK", "IIKW", "IWKI", "WIKI",
 "IIWK", "IWIK", "WIIK"]
```

Exercice 14

1) Que renvoie le programme `Anagrammes(L)` si tous les éléments de `L` sont distincts ?

2) On suppose que `L` n'a que des éléments distincts. Écrire un programme `Arrangements(L, k)` qui renvoie la liste de tous les arrangements à k éléments de `L`.

3) Traiter le cas général, où l'on ne suppose plus les éléments de `L` distincts. Il ne doit pas y avoir de répétitions dans la liste des résultats.

```
>>> Arrangements([1, 1, 2, 3], 3)
[[1, 1, 2], [1, 2, 1], [2, 1, 1], [1, 1, 3], [1, 3, 1], [3, 1, 1],
 [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Exercice 15 — On donne la fonction ci-dessous

```
def TousSauf(L, IndicesÀÉviter) :
    I = sorted(IndicesÀÉviter)
    p = 0
    for i in range(len(L)) :
        if p < len(I) and i == I[p] :
            p += 1
        else :
            yield L[i]
```

qui s'utilise comme un `range`, c'est-à-dire (par exemple) après un `in` dans une boucle `for`. Tester le programme ci-dessous, et en déduire ce que fait la fonction `TousSauf`.

```
def Test() :
    L = list("abcdefghijklmnopqrstuvwxyz")
    for x in TousSauf(L, [0, 4, 8, 14, 20, 24]) :
        print(x)
```

Exercice 16 — Déduire de ce qui précède une nouvelle version de `Arrangements(L)`, dans le cas où `L` n'a que des éléments distincts.