Le compte est bon ! (programmation dynamique et mémoïsation)

1 Principe

Dans ce jeu, on suppose disposer d'un ensemble E de n entiers positifs a_i ($0 \leqslant i < n$) et d'un entier positif p que l'on qualifiera d'*objectif*. Le but est de trouver comment il est possible de combiner tout ou partie de ces n entiers avec les quatres opérations usuelles (addition, soustraction, multiplication et division) de manière à obtenir l'objectif p, chaque entier étant utilisé au plus une seule fois.

Par exemple, pour $E = \{1,4,5,8,9\}$ et $p = 42$, une solution possible est $(1+5) \times (9 - 8/4)$. Une autre solution, n'utilisant pas tous les entiers, est $9 \times 5 - 4 + 1$.

On supposera que tous les résultats intermédiaires doivent être des entiers positifs ou nuls (il n'est donc pas permis, durant le calcul, d'effectuer une opération telle que $8/3$ ou $5-8$).

2 Résolution du problème

Un ensemble E d'entiers sera représenté en Python par un n-uplet (un « `tuple` » en Python), qui se comporte comme une liste, mais n'est pas mutable. On peut donc accéder à un élément en position `i` dans un tuple `t` en écrivant `t[i]` (mais il 'est pas possible de le modifier). Il n'est pas possible non plus d'utiliser les méthodes pop ou append avec un tuple, mais on peut concaténer deux tuples avec `+`. On peut transformer une liste `lst` en tuple en écrivant `tuple(lst)` (et inversement un tuple `t` en liste avec `list(t)`). Pour écrire un tuple, on sépare les éléments par des virgules, comme pour des listes, mais on ne les entoure pas de crochets. On utilise régulièrement des parenthèses cependant pour éviter tout problème de priorité : « `(37, 42, 54)` ». Un tuple à un seul élément s'écrit avec une virgule terminale (éventuellement entre parenthèses) : « `(42,)` ».

On se propose dans un premier temps d'écrire quelques fonctions qui nous seront utiles par la suite.

1. Proposer une fonction `pop2(tpl, i, j)` qui prend en argument un n-uplet de taille $n \geqslant 2$ et deux entiers distincts i et j dans $[\![0\,..\,n-1]\!]$ et retourne un triplet consistant en l'élément du n-uplet à la position i, celui à la position j, et un n-uplet de longueur $n-2$ contenant les éléments restants. Par exemple :

```
In []: pop2((2, 3, 5, 7, 7, 11), 3, 1)
Out[]: 7, 3, (2, 5, 7, 11)
```

2. Proposer une fonction `somme(tpl, i, j)` qui prend en argument un n-uplet de taille $n \geqslant 2$ et deux entiers distincts i et j dans $[\![0\,..\,n-1]\!]$, et retourne un n-uplet de taille $n-1$ où les deux éléments indiqués ont été remplacés par leur somme (placée en ernière position).

Par exemple,

```
In []: somme((2, 3, 5, 7, 7, 11), 3, 1)
Out[]: (2, 5, 7, 11, 10)
```

3. Proposer, sur le même principe, une fonction `produit(tpl, i, j)` :

```
In []: produit((2, 3, 5, 7, 7, 11), 3, 1)
Out[]: (2, 5, 7, 11, 21)
```

4. Pour la différence, comme l'opération n'est pas commutative, on construira une fonction `difference(tpl, i, j)` qui choisira systématiquement l'ordre, pour les deux éléments extraits, qui donne un résultat positif ou nul :

```
In []: différence((2, 3, 5, 7, 7, 11), 3, 1)
Out[]: (2, 5, 7, 11, 4)

In []: différence((2, 3, 5, 7, 7, 11), 1, 3)
Out[]: (2, 5, 7, 11, 4)
```

5. Enfin, proposer une fonction `quotient(tpl, i, j)` qui, si un des deux éléments est un multiple de l'autre, remplacera les deux éléments par leur quotient, **et retournera `None` si aucun des deux éléments n'est un multiple de l'autre** :

```
In []: quotient((2, 3, 5, 7, 7, 11), 3, 1)

In []: quotient((0, 6, 9, 2, 7, 7), 1, 3)
Out[]: (0, 9, 7, 7, 3)

In []: quotient((0, 6, 9, 2, 7, 7), 3, 1)
Out[]: (0, 9, 7, 7, 3)

In []: quotient((0, 6, 9, 2, 7, 7), 2, 0)
Out[]: (6, 2, 7, 7, 0)

In []: quotient((0, 6, 9, 2, 7, 7), 4, 5)
Out[]: (0, 6, 9, 2, 1)
```

Le premier appel ayant retourné `None`, cela apparait comme une absence de résultat dans l'interpréteur.

Pour savoir s'il est possible d'atteindre l'objectif, nous allons écrire une fonction récursive `possible(tpl, p)` qui devra retourner un booléen indiquant s'il est possible d'obtenir p à partir du n-uplet founit. Elle fonctionnera de la façon suivante :

- si p figure dans le n-uplet, elle retournera **`True`** ;
- sinon, pour tout couple d'entiers i et j vérifiant $0 \leqslant i < j < n$ (n étant la taille du n-uplet), on utilise chacune des quatres fonctions précédentes pour construire un (éventuel) n-uplet de taille $n-1$, on appelle la fonction `possible` sur cet n-uplet plus court, et si l'appel récursif retourne **`True`**, alors la fonction retourne **`True`** ;
- si tous les essais échouent, on retourne **`False`**.

6. Grâce à la démarche précédente, proposer une fonction `possible(tpl, obj)` prenant en argument un n-uplet `tpl` d'entiers et un entier `obj` représentant l'objectif à atteindre et retournant **`True`** ou **`False`** selon qu'il est possible ou non d'effectuer un calcul avec les entiers de E conduisant au résultat `obj`.

7. Vérifier avec la fonction écrite qu'il est possible d'obtenir, à partir des entiers $E = \{1,4,5,8,9\}$, les valeurs 42, 100 ou 199 mais pas 142. Combien de valeurs dans $[\![1\,..\,200]\!]$ ne peut-on pas atteindre ?

8. Proposer une fonction `possible_tous(tpl, obj)` qui effectue le même travail que `possible`, mais en altérant la terminaison pour qu'elle ne retourne **`True`** que si *tous* les entiers de E sont utilisés dans le calcul. Il y a *très peu* de choses à modifier par rapport à la fonction `possible` ! Combien de valeurs dans $[\![1\,..\,200]\!]$ ne peut-on pas atteindre avec $E = \{1,4,5,8,9\}$?

Dans la suite, on revient au cas où il n'est pas nécessaire d'utiliser tous les entiers de E**.**

9. Proposer une fonction `solution(tpl, obj)` procédant de la même manière que `possible` mais retournant une liste de chaînes de caractères décrivant les opérations à effectuer pour atteindre `obj`. La fonction retournera **`None`** si elle ne trouve aucune solution (et pourra retourner une liste vide si `obj` figure parmi les éléments de E). Par exemple, on pourra avoir :

```
In []: solution([1, 4, 5, 8, 9], 42)
Out[]: ['4+1 = 5', '8-5 = 3', '5+9 = 14', '14*3 = 42']

In []: solution([1, 4, 5, 8, 9], 142)
```

Le second appel a retourné **`None`** (et l'interpréteur n'a donc rien affiché) car il n'y avait pas de solution. Notons que la fonction peut parfois retourner des listes où certaines opérations ne sont pas indispensables au calcul du résultat final. On ne cherchera pas à les éliminer.

On souhaite trouver un ensemble E de quatre entiers distincts dans $[\![1\,..\,9]\!]$ tel qu'il soit possible d'obtenir tous les objectifs dans $[\![1\,..\,65]\!]$. Pour ce faire, on va tester tous les quadruplets, et regarder si `solution` [1] retourne une solution pour tout objectif dans $[\![1\,..\,65]\!]$. Pour se faciliter la vie, on dispose d'une fonction `combinations` dans le module `itertools` qui permet d'obtenir tous les quadruplets dont on a besoin. La fonction s'importe avec :

```
from itertools import combinations
```

Pour afficher par exemple tous les quadruplets, on peut écrire [2] :

```
for tpl in combinations(range(1, 10), 4):
    print(tpl)
```

10. Trouver l'unique quadruplet qui convient.

On souhaite, de même, trouver un quintuplet d'entiers distincts de $[\![1\,..\,9]\!]$ qui permette d'obtenir tous les entiers dans $[\![1\,..\,260]\!]$.

11. Essayer avec la fonction précédente. Que constate-t-on ?

3 Amélioration par mémoïsation

On souhaiterait gagner du temps. Si l'on regarde ce qui se passe, on effectue beaucoup de fois les mêmes calculs : si à partir de l'ensemble $\{1,2,3,4,5\}$ on calcule d'abord $1+2$ puis $3+4$, ou bien si l'on calcule d'abord $3+4$ puis $1+2$, dans les deux cas on se retrouve avec l'ensemble $\{3,5,7\}$. Il serait regrettable de vérifier *deux fois* qu'il est impossible d'atteindre l'objectif avec ces trois valeurs.

Pour ce faire, nous allons utiliser la mémoïsation, en créant un dictionnaire `mem` initialement vide, défini à l'extérieur de toute fonction, dont les clés seront un couple formé d'un n-uplet d'entiers et d'un objectif, soit les paramètres de la fonction `solution`, et les valeurs le résultat retourné par la fonction `Solution` (soit **`None`**, soit une liste d'opérations).

1. Proposer une fonction `solution_memo` pour que, si les paramètres correspondent à une clé de `mem`, elle retourne la valeur associée sans faire aucun calcul. Dans le cas contraire, elle effectue les calculs comme la fonction `solution`, mais mémorise le résultat dans `mem` avant de le retourner.

2. Vérifier que la fonction donne un résultat correct pour $\{1,4,5,8,9\}$ et les objectifs 42, 100 et 142.

3. Se servir de la fonction `solution_memo` pour identifier le quintuplet d'entiers souhaité.

En fait, on peut améliorer un peu les choses : les n-uplets `(2, 3, 5, 7)` et `(3, 5, 2, 7)` correspondent en fait à la *même* situation mathématique. Il serait intéressant, dans `mem`,

1. On peut également utiliser `possible`.
2. Le premier élement est l'ensemble des valeurs que l'on considère, $[\![1\,..\,9]\!]$ dans notre cas, et le second la taille des n-uplets souhaités, ici des quadruplets. L'itération produit des objets de type `tuple`.

de mémoriser des n-uplets *ordonnés*. On peut obtenir un n-uplet trié à partir d'un n-uplet quelconque `tpl` en écrivant `tuple(sorted(tpl))`.

4. Modifier la fonction `solution_memo` pour qu'elle stocke les n-uplets dans `mem` sous la forme de n-uplets *triés*. Attention, quand la fonction analysera ses arguments, elle devra également trier les éléments du n-uplet avant de les comparer aux clés de `mem` !

5. Vérifier que la fonction `solution_memo` peut trouver une solution pour atteindre 1234567 à partir de $E = \{3, 7, 11, 14, 17, 21, 22, 29, 42, 54, 78\}$ en un temps raisonnable, alors qu'il faut un temps considérable à `solution`.

6. Dans l'épreuve « Le compte est bon » du jeu télévisé « Des chiffres et des lettres », on tire six entiers et un objectif entre 1 et 999 (inclus). Vérifier avec les outils développés que le tirage $\{3, 4, 5, 9, 75, 100\}$ permet d'atteindre *tous* les objectifs possibles.