TP #4 — Deux petits exercices

Exercice 1. *Petite introduction aux bases de données*

Dans cet exercice, on souhaite manipuler des *données* stockées dans des *tables* . Les données d'une table sont ici représentées par des `tuple` distincts deux-à-deux et de même longueur ; une table est quant à elle représentée par une `list` contenant l'ensemble de ses données. Étant donnée une donnée, on utilisera alternativement le mot *colonne* pour désigner les indices du tuple représentant celle-ci ; c'est à dire que la valeur de la donnée d en la colonne i est donnée par `d[i]`.

On donne ci-dessous deux exemples de tables ainsi représentées en Python :

```
mustelidae = [("mustela nivalis","belette", "mustela"),
        ("mustela erminea","hermine", "mustela"),
        ("martes martes","martre des pins", "martes"),
        ("martes zibellina","zibeline", "martes"),
        ("lutra lutra","loutre d'eurasie", "lutra")]
ménagerie = [("aldo", "belette"), ("herminien", "hermine"),
        ("titus", "ours"), ("heide", "zibeline"),
        ("albertine", "jaguar")]
```

1. Écrivez une fonction `select1(table, cond)` qui prend en entrée une table *table* non vide (c'est à dire une liste non vide de tuples non vides) et une condition *cond* qui est un tuple d'un indice *ci* et d'une valeur *cv* et qui renvoie une table (c'est à dire une liste de tuples) contenant toutes les données t de *table* tels que la valeur de t à l'indice *ci* est égale à *cv*.
   Vous pouvez supposer ici que *ci* est toujours un indice valide pour les tuples contenus dans *table*.
   Par exemple, cette fonction doit renvoyer les résultats suivants :

```
> select1(mustelidae, (2, "mustela"))
  [('mustela nivalis', 'belette', 'mustela'),
   ('mustela erminea', 'hermine', 'mustela')]
> select1(mustelidae, (1, "loutre"))
  []
> select1(mustelidae, (1, "loutre d'eurasie"))
  [('lutra lutra', "loutre d'eurasie", 'lutra')]
```

On souhaite généraliser la fonction `select1` de deux façons : on veut pouvoir préciser plus d'une condition, c'est à dire contraindre les valeurs des données sur plusieurs de leurs indices ; on veut pouvoir renvoyer des données qui sont des tuples « plus courts » que les données originales, en gardant seulement les valeurs correspondant à certaines colonnes (c'est à dire certains indices).

2. Écrivez une fonction `select(cols, table, conds)` qui prend en entrée une table *table* comme ci-dessus, un tuple *cols* d'indices valides distincts pour les données de *table*, un tuple *conds* **possiblement vide** de tuples *(ci,cv)* d'indices (valides pour les données de *table*) et de valeurs, et qui renvoie une table contenant les données de *table* vérifiant toutes les conditions dans *conds*, dont on a seulement gardé les valeurs en les colonnes données par *cols*.
Par exemple, cette fonction doit renvoyer les résultats suivants :

```
> select((1,), mustelidae, ((2,"mustela"),))
  [('belette',), ('hermine',)]
> select((0,1), mustelidae, ((2,"mustela"),))
  [('mustela nivalis', 'belette'),
  ('mustela erminea', 'hermine')]
> select((0,1), mustelidae, ((2,"mustela"),
         (1,"belette")))
  [('mustela nivalis', 'belette')]
```

Rappel : étant donnés deux tuples *t1* et *t2*, leur concaténation est donnée par `t1 + t2`.

On s'intéresse enfin au problème de calculer une *jointure* de deux tables, c'est à dire le produit cartésien des données des deux tables où l'on garde seulement les (concaténation des) données qui sont égales sur un ensemble de leurs colonnes.

3. Écrivez une fonction `join(table1, table2, jointcols)` où *table1* et *table2* sont deux tables (de nombre de colonnes non nécessairement identiques) et *jointcols* un tuple de couples d'indices *(i1, i2)* qui sont des indices de colonnes valides pour la première et seconde table respectivement, et qui renvoie une table contenant les tuples *t1 + t2* des données du produit cartésien de *table1* et *table2* tels que `t1[i1]` et `t2[i2]` sont égaux pour les couples *(i1, i2)* de *jointcols*.
Par exemple, cette fonction doit renvoyer les résultats suivants :

```
> join(mustelidae, ménagerie, ((1,1),))
  [('mustela nivalis', 'belette', 'mustela',
    'aldo', 'belette'),
   ('mustela erminea', 'hermine', 'mustela',
    'herminien', 'hermine'),
   ('martes zibellina', 'zibeline', 'martes',
    'heide', 'zibeline')]
> join(mustelidae, ménagerie, ((0,1),))
  []
```

On remarque par ailleurs que cette fonction peut être composée avec `select` pour par exemple donner :

```
> select((3,1,0),
         join(mustelidae, ménagerie, ((1,1),)), ())
```

```
[('aldo', 'belette', 'mustela nivalis'),
 ('herminien', 'hermine', 'mustela erminea'),
 ('heide', 'zibeline', 'martes zibellina')]
```

4. Quel est le coût pire cas de votre fonction `join` ? Celle-ci vous paraît-elle être améliorable ?

On souhaite utiliser un dictionnaire pour améliorer les performances de `join` dans le cas où la jointure ne se fait que relativement à un unique (couple d') indice. L'idée est la suivante : pour calculer `join(t1, t2, ((i1, i2),))`, on commence par construire un dictionnaire pour l'une des tables `t1` ou `t2` (disons `t1`) qui « indexe » ses données par rapport à leurs valeurs suivant l'indice `i1` : la valeur du dictionnaire associée à la clef `k` contient la liste (éventuellement vide) des données de `t1` valant `k` à l'indice `i1`. Ce dictionnaire est ensuite utilisé pour accélérer le calcul de la jointure proprement dite.

5. Écrivez une fonction `join2` implémentant l'approche décrite ci-dessus.
6. Testez.
7. Proposez (sans nécessairement l'implémenter) une autre approche (qui n'utilise pas de dictionnaire) permettant d'éventuellement accélérer le calcul d'une jointure suivant un unique (couple d') indice.

Exercice 2. *Rendu de monnaie dynamique*

On s'intéresse dans cet exercice au classique *problème du rendu de monnaie* : soit un entier strictement positif k et un tuple B de valeurs de billets `B[i]` qui sont elles-mêmes des entiers strictement positifs, on souhaite trouver un tuple ou une liste C satisfaisant les trois conditions suivantes :

i. tout élément de C est un élément de B
ii. la somme des éléments de C est égale à k
iii. la longueur de C (c'est à dire son nombre d'éléments) est minimale parmi tous les tuples C satisfaisant les deux premières conditions.

Par exemple, si B vaut `(1,4,7,13,28,52,91,365)` et k vaut `5` :

— le tuple `(5)` satisfait la seconde condition mais pas la première
— le tuple `(1,1,1,1,1)` satisfait les deux premières conditions mais pas la troisième
— le tuple `(1,4)` satisfait les trois conditions et est donc une réponse admissible à cette instance du problème.

On rappelle qu'une *stratégie gloutonne* pour tenter de résoudre ce problème consiste à itérativement construire C comme une liste en y ajoutant la plus grande valeur de B encore possible (c'est à dire, qui fait que la somme des éléments de C ne dépassera pas k). Par exemple, pour k valant `21` et B comme dans l'exemple précédent, cette stratégie construit itérativement C comme `[13]`, `[13, 7]` et enfin `[13, 7, 1]`.

Dans toute la suite de cet exercice, on fixe le tuple B à la valeur de l'exemple (empruntée à Jeff Erickson), c'est à dire que B vaut `(1,4,7,13,28,52,91,365)`.

1. Écrivez une fonction récursive `changeOpt` de même spécifications qu'à la question précédente. qui résout le problème du changement de monnaie pour son argument k et B fixé comme ci-dessus en renvoyant un tuple C satisfaisant les trois conditions ci-dessus. Cette fonction n'a pas besoin d'être efficace, mais elle doit être correcte.
2. Écrivez une variante `changeOptM` de votre fonction `changeOpt` qui utilise la mémoïsation pour être plus efficace que cette dernière.
3. Écrivez une fonction `notGreed()` qui ne prend aucun argument et qui :
   a. si elle termine, retourne un entier k pour lequel la stratégie gloutonne rappelée ci-dessus ne résout pas le problème (c'est à dire qu'elle ne donne pas un tuple C de longueur minimale) ;
   b. termine s'il existe un tel entier k.