Méthode de Mémoïsation

On se place dans le cadre d'un problème donné, $\mathcal{P}(e_1, \cdots, e_n)$ et de ses sous-problèmes que l'on notera $\mathrm{P}(e_1, \cdots, e_n, k_1, \cdots, k_m)$ avec $k_1, \cdots, k_m$ des entrées supplémentaires *immutables*.

On suppose qu'on connaît une définition par récurrence de P :

$$\begin{cases} \mathrm{P}(e_1, \cdots, e_n, i_1^{(1)}, \cdots, i_m^{(1)}) & = a_1 \\ \vdots & \vdots \\ \mathrm{P}(e_1, \cdots, e_n, i_1^{(r)}, \cdots, i_m^{(r)}) & = a_r \\ \mathrm{P}(e_1, \cdots, e_n, k_1, \cdots, k_m) & = f(\mathrm{P}(e_1, \cdots, e_n, k_1^{(1)}, \cdots, k_m^{(1)}), \cdots, \mathrm{P}(e_1, \cdots, e_n, k_1^{(p)}, \cdots, k_m^{(p)})) \end{cases}$$

Avec $a_1, \cdots, a_r$ les r valeurs des cas initiaux et f une fonction qui permet d'obtenir une sous problèmes à partir de p sous problèmes plus petits.

On connait aussi $q_1, \cdots, q_m$ les entrées telles que :

$$\mathcal{P}(e_1, \cdots, e_n) = \mathrm{P}(e_1, \cdots, e_n, q_1, \cdots, q_m)$$

I Approche Descendante du Calcul

L'approche descendante du calcul s'implémente à l'aide d'une fonction récursive qui copie la définition précédente.

```
def P(e_1, ..., e_n, k_1, ..., k_n) :
    if (k_1, ..., k_n) == (i_1, ..., i_n) :
        return a_1
    ⋮
    if (k_1, ..., k_n) == (i_1, ..., i_n) :
        return a_r
    v_1 = P(e_1, ⋯, e_n, k_1^(1), ⋯, k_m^(1))
    ⋮
    v_p = P(e_1, ⋯, e_n, k_1^(p), ⋯, k_m^(p))
    return f(v_p, ..., v_p)

def 𝒫(e_1, ⋯, e_n) :
    return P(e_1, ⋯, e_n, q_1, ⋯, q_m)
```

II Mémoïsation

Dans le cadre où il y a *dépendance des sous-problèmes* on peut mémoïser la fonction précédente.

On choisi une structure de mémoïsation permettant d'associer à des entrées, les valeurs de la fonction. Il nous faut une structure de tableau associatifs (liste, matrices, arbre, graphe, dictionnaire,…). Dans le doute un dictionnaire fonctionnera toujours.

La mémoïsation se fait selon l'algorithme suivant (à adapter si on n'utilise pas un dictionnaire) :

1. Ajouter un paramètre `memo` de type dictionnaire à la fonction `P` de calcul des sous-problème *ainsi qu'à tous ses appels récursifs*.
2. Ajouter une conditionnelle en préambule de la fonction `P` qui, si les entrées $(k_1, \ \ldots, \ k_n)$ sont déjà une clé de `memo` renvoie la valeur associée à ces entrées, et ne fait rien sinon.
3. Avant chaque retour de la fonction, ajouter dans `memo` l'association de clé $(k_1, \ \ldots, \ k_n)$ et de valeur ce qui est renvoyé. On remplace le retour par le retour de `memo[`$(k_1, \ \ldots, \ k_n)$`]`.
4. On ajoute un dictionnaire vide en entrée de l'appel à `P` dans la définition de $\mathcal{P}$.

Voilà le résultat.

```
def P(e_1, ..., e_n, k_1, ..., k_n, memo) : # (1)
    if (k_1, ..., k_n) in memo : # (2)
        return memo[(k_1, ..., k_n)]
    if (k_1, ..., k_n) == (i_1, ..., i_n) :
        memo[(k_1, ..., k_n)] = a_1 # (3)
        return memo[(k_1, ..., k_n)]
    ⋮
    if (k_1, ..., k_n) == (i_1, ..., i_n) :
        memo[(k_1, ..., k_n)] = a_r # (3)
        return memo[(k_1, ..., k_n)]
    v_1 = P(e_1,⋯,e_n,k_1^(1),⋯,k_m^(1), memo) # (1)
    ⋮
    v_p = P(e_1,⋯,e_n,k_1^(p),⋯,k_m^(p), memo) # (1)
    memo[(k_1, ..., k_n)] = f(v_p, ..., v_p) # (3)
    return memo[(k_1, ..., k_n)]

def 𝒫(e_1,⋯,e_n) :
    return P(e_1,⋯,e_n,q_1,⋯,q_m, {}) # (4)
```