TP 08 – Corrigé

Algorithmes gloutons

Activité 1 – Algorithme glouton du rendu de monnaie

La société Sharp commercialise des caisses automatiques utilisées par exemple dans des boulangeries. Le client glisse directement les billets ou les pièces dans la machine qui se charge de rendre automatiquement la monnaie.

Objectif Afin de satisfaire les clients, on cherche à déterminer un algorithme qui va permettre de rendre le moins de monnaie possible.

La machine dispose de billets de 20€, 10€ et 5€ ainsi que des pièces de 2€, 1€, 50, 20, 10, 5, 2 et 1 centimes.

On se propose donc de concevoir un algorithme qui demande à l'utilisateur du programme la somme totale à payer ainsi que le montant donné par l'acheteur. L'algorithme doit alors déterminer quels sont les billets et les pièces à rendre par le vendeur.

Pour ne pas faire d'erreurs d'approximation, tous les calculs seront faits en centimes.

Le contenu de la caisse automatique et le contenu du porte-monnaie du client seront modélisés par un tableau ayant la forme suivante :

```
caisse = [[2000,5],[1000,5],[500,5],[200,5],[100,5],[50,5],[20,5],[10,5],[5,5],[2,5],[1,5]]
```

Cela signifie que la caisse contient 5 billets de 20€, 5 billets de 10€...

Dans la prochaine question, on fait l'hypothèse que la caisse contient suffisamment de billets et de pièces de chaque valeur.

Question 1 *Ecrire une fonction* `rendre_monnaie(caisse:list, cout:float, somme_client:float)->list` *prenant en arguments deux flottants* `cout` *et* `somme_client` *représentant le coût d'un produit et la somme donnée par le client en€ ainsi que le contenu de la caisse. Cette fonction renvoie la liste des billets à rendre par le client.*

Ainsi, l'instruction `rendre_monnaie(caisse, 16, 20)` renvoie la liste `[200, 200]`.
`rendre_monnaie(caisse,15.92,20)` renvoie `[200, 200, 5, 2, 1]`

Question 2 *Ecrire une fonction* `rendre_monnaie_v2(caisse:list, cout:float, somme_client:float)->list` *ayant le même objectif que la précédente. Cette fonction devra de plus mettre à jour la caisse. Elle devra prendre en compte que la caisse peut manquer de billets. Elle renverra une liste vide s'il n'est pas possible de rendre la monnaie.*

Pour l'instruction `rendre_monnaie_v2(caisse, 15.99, 200)`, la liste renvoyées est la suivante : `[2000, 2000, 2000, 2000, 2000, 1000, 1000, 1000, 1000, 1000, 500, 500, 500, 500, 500, 200, 200, 200, 200, 100, 1]`. Le contenu de la caisse est alors : `[[2000, 0], [1000, 0], [500, 0], [200, 1], [100, 4], [50, 5], [20, 5], [10, 5], [5, 5], [2, 5], [1, 4]]`.

L'instruction `rendre_monnaie_v2(caisse, 15.99, 200)` renvoie `[]`.

On suppose que la caisse est maintenant la suivante.

```
caisse = [[5000,10],[2000,10],[1000,10],[800,10],[100,10]]
```

Le client achète un article de 34€ avec un billet de 50€.

Question 3 *Que retourne la fonction* `rendre_monnaie` *? Est-ce le rendu optimal ? Conclure « qualitativement ».*

Pour l'instruction `rendre_monnaie_v2(caisse, 34, 50)`, l'algorithme renverra `[1000, 100, 100, 100, 100, 100, 100]`.

Activité 2 – Recherche d'un plus court chemin par un algorithme glouton

Résolution du problème

Question 4 *Écrire la fonction* `distance(p1:list,p2:list)-> float` *permettant de calculer la distance euclidienne entre deux points. Vous importerez les fonctions que vous jugerez utile. La fonction* `test_Q4()` *permet de valider votre fonction dans un cas.*

Correction

```
from math import sqrt
def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

def distance2(p1, p2): # sans import
    x1, y1 = p1
    x2, y2 = p2
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2)**0.5
```

Question 5 *Écrire la fonction* `distances(a:list,pts:list)-> list` *permettant de calculer l'ensemble des distances entre chacun des n points P de la liste* `pts` *et chacune des distances entre le point* `a` *et chacun des points de la listes* `pts` *conformément à l'exemple ci-dessus.*

Correction

```
def distances(pts, dep):
    n = len(pts)
    tab = [(n+1)*[0] for i in range(n+1)]
    for i in range(n):
        for j in range(i):
            tab[i][j] = distance(pts[i], pts[j])
            tab[j][i] = tab[i][j]
        tab[n][i] = distance(dep, pts[i])
        tab[i][n] = tab[n][i]
    return tab
```

Question 6 *En prenant compte de la note précédente, estimer le nombre de distances à calculer par l'algorithme.*

Correction Le tableau des distances contient $n+1$ lignes et colonnes soit $(n+1)\times(n+1)$ valeurs.
Les termes de la diagonale sont tous nuls. Il ne faut donc pas les calculer. Il y en a $n+1$.
Le tableau est symértrique ce qui signifie que `tab[i][j]=tab[j][i]`.
Il faut donc calculer $\dfrac{(n+1)^2-(n+1)}{2}=\dfrac{n(n+1)}{2}$ distances.

Soit un chemin, quelconque passant par un ensemble de points contenu dans le tableau des distances. Pour modéliser ce chemin, on utilise la liste des index des points qui le constituent. Ainsi si `chemin = [n,4,0,n-1]` alors c'est qu'on a parcouru le chemin $A \to P_5 \to P_1 \to P_n$.

Question 7 *Écrire la fonction* `longueur(chemin:list, tab:list) -> float` *où* `tab` *est un tableau de distances déterminé par la fonction* `distances`.

Correction

```
def longueur(chemin, dist):
    d = 0
    id_pt = len(dist) - 1
    for point in chemin:
        d = d + dist[id_pt][point]
        id_pt = point
    return d
```

Question 8 *Proposer une version récursive de cet algorithme. On la notera* `longueur_rec(chemin:list, tab:list) -> float`.

Correction

```
def longueur(chemin, dist): A TESTER !!
    d = 0
    if len(chemin) == 2 :
        return dist[chemin[0][chemin[1]]
    else :
        return d=d + longueur(chemin([1:],dist))
```

Question 9 *On considère que* `dispo=[True, True, False, False, True, False]`. *Combien existerait-il de points dans la variable* `pts` *définie précédemment ? Combien de points ont été parcouru ? Citer les points visités.*

Correction

`pts` est constitué de 5 points. 3 points ont été visités : A, P_3 et P_4.

Question 10 *Dans la figure ci-dessus, expliquer pourquoi il y a une case à ne pas visiter ?*

Correction On ne visite pas la case où on est ...

Question 11 *Écrire une fonction* `indice(i:int, tab:list, dispo:list)->int` *permettant de déterminer l'index du point le plus proche du point d'index* `i`, *parmi les points disponibles.* `tab` *désigne le tableau des distances créé avec la fonction* `distances`.

Correction

```
def indice(position, dist, dispo):
    n = len(dist) - 1
    global dim
    mini = 3 * dim # supérieur àla diagonale du carré
    for i in range(n):
        if dispo[i]:
            d = dist[position][i]
            if d < mini:
                mini = d
                ind = i
    return ind
```

Question 12 *Écrire la fonction* `plus_court_chemin(dist:list)->list` *permettant de construire la chemin le plus court. Le chemin sera constitué de la liste des index des points. On initialisera donc le chemin avec le plus grand index de* `dist`. *On initialisera la liste* `dispo` *des points disponibles. Á chaque itération, on ajoutera à* `chemin` *le point le plus proche puis on mettra à jour la variable* `dispo`.

Correction

```
def plus_court_chemin(dist):
    n = len(dist) - 1
    chemin = []
    dispo = n * [True]
    position = n
    while len(chemin) < n:
        position = indice(position, dist, dispo)
        chemin.append(position)
        dispo[position] = False
    return chemin
```

Représentation du chemin

Question 13 *Tester le fonction* `plot_chemin()`. *Commenter l'ensemble des lignes en effectuant les regroupements vous paraissant nécessaires.*