Algorithmique:
Corrigé TP1: Greedy

1 Rendu de monnaie britannique

1.1 algorithme

Voici, en pseudocode, une version possible de l'algorithme greedy vu en cours pour le rendu de la monnaie :

```
Data: coin_set, int total
Result: None
begin
    while coin_set ≠ ∅ do
        val ← max s ∈ coin_set;
        if val ≤ total then
            total ← total − val;
            print(val);
        end
        else
            coin_set.remove(val);
        end
    end
end
```

Algorithm 1: *renduMonnaie*

1.2 Greedy

Cet algorithme appartient à la famille des algorithmes gloutons car c'est un algorithme d'optimisation (minimisation du nombre de pièces rendues) qui fait toujours des choix *localement* optimaux dans l'espoir que cela mène à une solution globalement optimale.

1.3 Optimalité

L'algorithme n'est pas toujours optimal : il suffit de prendre comme exemple 48 et voir que l'algorithme répond (30+12+6) alors qu'il existe une solution en 2 pièces (24+24).

1.4 Implémentation Python

Se référer au fichier *monnaiegreedy.py* disponible sur Moodle.

2 Espace disque

2.1 maximisation du nombre de fichiers

Ici nous prenons à chaque fois le fichier le plus petit parmi les fichiers restants. Montrons l'optimalité de l'algorithme. Si l'espace disque total a une taille D et s'il est rempli par les k plus petits fichiers, nous savons que :

$$\sum_{i=1}^{k} val(f_i) \leq D.$$

Créons un fichier supplémentaire f_f d'une taille de :

$$D - \sum_{i=1}^{k} val(f_i),$$

de sorte que :

$$val(f_f) + \sum_{i=1}^{k} val(f_i) = D,$$

et insérons-le dans la mémoire, qui devient maintenant pleine. Nous savons que $val(f_f) < val(f_{k+1})$ car sinon nous aurions pu mettre le fichier f_{k+1} dans la mémoire. Maintenant que la mémoire est completement pleine, nous savons que l'algorithme est optimal si et seulement si il est impossible de remplacer un nombre ω de fichiers choisis par l'algorithme par un nombre plus grand de fichiers non-choisis. Ceci n'est pas possible car les fichiers choisis sont bornés superieurement par $val(f_{k+1})$ et les délaissés sont bornés inférieurement par cette meme valeur. En effet, dans le pire des cas, le plus gros des fichiers choisis vaut $val(f_k) = val(f_{k+1})$. Formellement :

$$D + val(f_{k+1}) > D,$$

mais :

$$D + val(f_{k+1}) = \underbrace{\sum_{i=1}^{k} val(f_i) + val(f_f)}_{D} - \omega \cdot val(f_{k+1}) + (\omega + 1) \cdot val(f_{k+1}),$$

ce qui montre que l'algorithme est optimal.

2.2 minimisation de l'espace mémoire

Ici un algorithme greedy serait de prendre toujours le plus grand fichier possible, pour occuper un maximum de place avec chaque fichier. L'algo n'est pas optimal car si on considère une place mémoire de taille $N = 6$ et un ensemble de fichiers de taille $\{5, 2, 2, 2\}$ nous voyons que l'algo va prendre le fichier de taille 5 et avoir une espace restant de 1, alors que s'il avait choisi les 3 fichiers de taille 2 il aurait tout rempli. En fait, on ne peut donner d'algorithme greedy qui soit optimal pour la minimisation de l'espace mémoire ; quand bien même le problème ne semble de prime abord pas plus complexe que celui de la maximisation du nombre de fichier, il l'est en réalité.

3 Traversée du désert

3.1 Condition minimale

Il faut que la distance entre deux puits consécutifs soit toujours inférieure à d : $X_{i+1} - X_i \leq d, \forall i$

3.2 Premier cas

Il n'y a pas de solution possible dans cet exemple car entre 21 et 37 la différence est plus grande que d (10).

3.3 Algorithme

La fichier *desert.py* disponible sur Moodle fourni un code simple pour la résolution de ce problème.

3.4 Second cas

L'algorithme répond la suite 10, 20, 22, tandis que d'autres solutions en 3 étapes sont possibles comme par exemple 5, 12, 22 ou encore 10, 12, 22.

3.5 Optimalité

Notons $X_{i,a}$ le puits atteint par l'algorithme a à l'étape i. Nommons g l'algorithme greedy et c un algorithme concurrent, supposé meilleur que g tout en faisant, au moins une fois, un arrêt avant g, c'est-à-dire qu'il y a au moins une étape où il se différencie du greedy, en faisant le choix 'malin' de ne pas aller le plus loin possible. Si à l'étape i le concurrent est derrière g, nous avons :

$$X_{i,c} < X_{i,g}. \tag{1}$$

L'étape atteinte par g en $i+1$ est, par définition, le point p appartenant à l'ensemble des points X qui maximise la distance depuis la position à l'étape i :

$$X_{i+1,g} = \arg\max_{p} \left(p - X_{i,g} | p - X_{i,g} \leq d, p \in X\right). \tag{2}$$

D'un autre côté, la plus lointaine étape atteignable par c est :

$$X^{max}_{i+1,c} = \arg\max_{p} \left(p - X_{i,c} | p - X_{i,c} \leq d, p \in X\right). \tag{3}$$

En se servant de (**??**), on peut par ailleurs écrire :

$$p - X_{i,c} < p - X_{i,g} \ , \forall p \in X. \tag{4}$$

Ainsi, le passage dans l'arg max ne fait que changer le signe de l'inégalité, passant d'un 'strictement inférieur' à un 'inférieur ou égal', du fait de la contraite $p \in X$:

$$X^{max}_{i+1,c} \leq X_{i+1,g}.$$

Autrement dit, sur deux étapes, l'algorithme g prend *nécessairement* la plus grande valeur du set de valeurs permises entre $X_{i,g}$ et $X_{i,g} + d$, tandis que l'algorithme c ne peut pas dans tous les cas l'atteindre. Le non greedy est donc, au mieux, égal au greedy sur deux étapes.

On peut voir la généralisation à n étapes ainsi : pour que c parvienne avant g à la sortie du désert, il faut qu'il ait dépassé g au moins une fois. Or, localement, à chaque étape, il ne peut le faire, en vertu de ce que nous venons de montrer. Par conséquent, globalement, il ne peut le faire.