Algorithmes gloutons

I./ Problèmes d'optimisation :

Parmi tous les problèmes pouvant être résolus par de façon algorithmique, une catégorie importante est constituée des problèmes d'optimisation. Il s'agit de problèmes où l'objectif est de trouver la meilleure solution parmi un très grand nombre de solutions possibles. Trouver la meilleure solution revient à minimiser (ou maximiser) un critère.

Exemple : un GPS peut trouver le meilleur itinéraire entre deux villes. Le meilleur itinéraire sera différent selon que le critère à minimiser est la distance à parcourir ou le temps nécessaire pour le trajet.

Une façon simple pour résoudre ce type de problème est la méthode « force brute », qui consiste à recenser toutes les solutions possibles au problème (dans notre exemple toutes les routes possibles entre les deux villes), d'évaluer le critère à minimiser pour chacune de ces solutions, et de déterminer ainsi la solution donnant le critère minimal : avec cette méthode on est donc assuré de trouver la solution optimale. L'inconvénient de cette méthode est qu'elle peut prendre beaucoup de temps si le nombre de solutions à explorer est très grand.

Il existe une classe d'algorithmes utilisées pour déterminer en un temps raisonnable une solution acceptable (mais pas forcément optimale) à un problème d'optimisation complexe. On appelle ce type d'algorithmes une heuristique. Un exemple de ce type d'algorithmes est l'algorithme glouton.

II./ Algorithme glouton :

1./ Principe de base :

La résolution d'un problème se fait le plus souvent en plusieurs étapes. Le principe d'un algorithme glouton est de faire le choix qui semble le plus avantageux à chaque étape, sans tenir compte des choix qui ont été faits aux étapes précédentes. Nous allons illustrer ce principe sur deux exemples classiques.

2. Problème du sac à dos :

On dispose d'un sac à dos ayant une certaine contenance (par exemple 30 kg). On désire remplir ce sac avec des objets ayant chacun une masse et un prix :

Objet	**A**	**B**	**C**	**D**
Masse	**13 kg**	**12 kg**	**8 kg**	**10 kg**
Prix	**700 €**	**400 €**	**300 €**	**300 €**

Le but est de remplir le sac de façon à ce que son contenu ait le prix le plus grand possible.

La première chose à faire est de déterminer quel est l'objet le plus intéressant pour la résolution du problème : a priori c'est l'objet qui rapportera le plus d'argent, tout en pesant le moins, afin d'occuper le moins de place dans le sac. Il faut donc déterminer le prix par unité de masse (prix massique ?) pour chaque objet, ce que l'on obtient en divisant le prix en € par la masse en kg. On obtient alors les valeurs suivantes (arrondies à l'entier le plus proche) :

Objet	**A**	**B**	**C**	**D**
Prix massique	**54 €/kg**	**33 €/kg**	**38 €/kg**	**30 €/kg**

L'objet le plus intéressant à placer dans le sac est donc l'objet A, qui a le « prix massique » le plus intéressant. On a donc placé un objet de masse 13 kg dans notre sac pouvant en contenir 30. Il nous reste donc une capacité libre de 17 kg.

A la deuxième étape, le choix le plus intéressant est l'objet C, avec un « prix massique » de 38 €/kg. On la place donc dans le sac car sa masse est inférieure à la capacité restante. La capacité libre est alors de 17 – 8 = 9 kg.

On se rend compte à ce niveau là qu'il ne nous reste plus assez de place pour rajouter un des deux autres objets. L'algorithme s'arrête donc là : notre sac contient donc les objets A et C, pour un prix total de 700 + 300 = 1000 €.

Cette solution n'est pas optimale : en effet si on avait placé les objets A et B dans le sac, on aurait obtenu un prix total de 1100 €, pour un poids total de 25 kg. Mais le choix de l'objet B ne semblait pas être le plus pertinent lors de la deuxième étape.

Ecrire une fonction **`sac_glouton(objets, prix, poids, contenance_sac)`** qui attend en paramètres trois listes (une contenant le nom des objets, une contenant le prix des objets, la dernière contenant le poids des objets), et un entier correspondant à la contenance totale du sac. La fonction devra renvoyer : le contenu du sac déterminé par l'algorithme glouton décrit précédemment, le prix total des objets contenus dans le sac, ainsi que le poids total des objets contenus dans le sac.

Avec les données de l'exemple, on aurait :

```
>>> sac_glouton(['A', 'B', 'C', 'D'], [700, 400, 300, 300], [13, 12, 8, 10], 30)
(['A', 'C'], 1000, 21)
```

Ecrire une fonction **`sac_bourrin(objets, prix, poids, contenance_sac)`** qui attend en paramètres trois listes (une contenant le nom des objets, une contenant le prix des objets, la dernière contenant le poids des objets), et un entier correspondant à la contenance totale du sac. La fonction devra renvoyer : la même chose que la fonction précédente mais en utilisant la force brute : il faut donc déterminer toutes les combinaisons possibles d'objets qui rentrent dans le sac, et évaluer le prix total pour chacune de ses combinaisons afin de choisir celle qui donnera le prix total maximum.

3. Problème du rendu de monnaie :

On dispose d'une caisse contenant un nombre illimité de pièces de 2 et 1 euros, ainsi que des pièces de 1, 2, 5, 10, 20 et 50 centimes. Le but du problème est de rendre à un client une certaine somme de monnaie, en utilisant le moins de pièces possibles. Par exemple, si on doit rendre la somme de 4,87 €, la solution utilisant le moins de pièces est : 2 pièces de 2 €, 1 pièce de 50 centimes, 1 pièce de 20 centimes, 1 pièce de 10 centimes, 1 pièce de 5 centimes, et 1 pièce de 2 centimes, soit un total de 7 pièces.

L'algorithme glouton permettant de résoudre ce problème consiste à choisir la pièce ayant la plus grande valeur possible. Soit 4,87 € à rendre. La pièce ayant la valeur la plus importante est la pièce de 2 €. Il nous reste donc 4,87 – 2 = 2,87 € à rendre. La pièce ayant la plus grande valeur que l'on peut rendre est à nouveau la pièce de 2 €. Il nous reste 0,87 € à rendre. La pièce ayant la plus grande valeur que l'on peut rendre est à nouveau la pièce de 50 centimes. Il nous reste 0,37 € à rendre. Etc… On retrouve les pièces indiquées précédemment.

Ecrire une fonction **`rendu_monnaie(somme)`** qui attend une somme en euros et qui renvoie une liste contenant les pièces déterminées par l'algorithme décrit précédemment. Remarque : comme nous l'avons vu précédemment, on peut avoir des surprises lorsqu'on effectue des calculs avec des nombres flottants (essayez **`0.1 + 0.2`** pour voir !). Pour éviter ces complications, une méthode possible est en fait de travailler non pas en euros, mais en centimes : on ne manipulera alors que des nombres entiers.

Avec l'exemple précédent :

```
>>> rendu_monnaie(4.87)
[2.0, 2.0, 0.5, 0.2, 0.1, 0.05, 0.02]
```

On peut démontrer qu'avec le choix de pièces utilisées (2, 1, 0.50, 0.20, 0.10, 0.05, 0.02, 0.01), la solution trouvée par l'algorithme glouton sera toujours optimale. Ce ne serait pas le cas avec un autre jeu de pièces : essayer de tester votre algorithme avec le jeu de pièces suivant : 1, 3, 6, 12, 24 et 30, et faites-lui rendre la somme de 49 €. La solution proposée est-elle optimale ?