Rendu de monnaie

Thème des algorithmes gloutons

1ère NSI

Plan du chapitre

Plan du chapitre

Quand un vendeur rend la monnaie a un client, peut-on systématiquement trouver une manière de rendre la monnaie qui minimise le nombre de pièce et de billets ?

La même phrase en langage informatique : « existe-t-il une solution optimale au rendu de monnaie ? »

Quand un vendeur rend la monnaie a un client, peut-on systématiquement trouver une manière de rendre la monnaie qui minimise le nombre de pièce et de billets ?

La même phrase en langage informatique : « existe-t-il une solution optimale au rendu de monnaie ? »

Il y a une hypothèse cachée derrière cet énoncé. Laquelle ?

Quand un vendeur rend la monnaie a un client, peut-on systématiquement trouver une manière de rendre la monnaie qui minimise le nombre de pièce et de billets ?

La même phrase en langage informatique : « existe-t-il une solution optimale au rendu de monnaie ? »

Il y a une hypothèse cachée derrière cet énoncé. Laquelle ?

Le jeu de monnaie
(la valeur des pièces et des billets)

La réponse à la question de la solution optimale du rendu de monnaie avec un jeu de monnaie quelconque est un problème `NP-Complet` qui est l'un des 7 problèmes du millénaire : `https://fr.wikipedia.org/wiki/Probl%C3%A8mes_du_prix_du_mill%C3%A9naire`

Dans le cas où le jeu de monnaie est le jeu européen actuel (le choix des valeurs des pièces et des billets n'a pas été fait au hasard), la **solution gloutonne est optimale**.

Plan du chapitre

Définition

Un algorithme glouton est un algorithme qui fait un **choix local optimal** en espérant que cela conduise à une **solution globale optimale**.

Un algorithme glouton ne conduit pas toujours à une solution optimale.

Un algorithme glouton fait toujours le choix qui lui semble le meilleur sur le moment.

Un algorithme glouton ne remet jamais en cause son choix.

Plan du chapitre

Nous sommes des commerçants et nous avons des pièces/billets de

- 1 €
- 2 €
- 5 €
- 10 €
- 20 €
- 50 €
- 100 €
- 200 €
- 500 €

en nombre illimité (on peut toujours rêver !)

Travail 1

1. Trouver toutes les possibilités de rendre 7 € en monnaie.
2. Existe-t-il une ou plusieurs possibilités optimales en nombre de pièce (le moins de pièce/billet possible) ?

Travail 2

1. Trouver une manière gloutonne de rendre la monnaie.
2. Appliquer votre procédure à la somme de 263 €. Combien de pièce avez-vous utiliser ?

Travail 3 : écrire un algorithme glouton de rendu de monnaie

```
# coding: utf-8
valeurs = [1,2,5,10,20,50,100,200,500] # en euros
def rendu_monnaie_glouton (valeurs, montant) :
    """
    étant donné un jeu de pièces 'valeurs'
    renvoie une liste 'pieces' correspondant
    au nombre nécessaire de chaque pièce de monnaie
    """


    return pieces
pieces = [1,0,1,0,0,0,0,0,0] # 1€ + 5€ (exemple)
```

En vue de l'année prochaine :

Travail 4 : Ecrire une version récursive de votre algorithme glouton du rendu de monnaie.
Un rappel sur la récursivité est présenté pages suivantes.

Par la **récursivité**, on détermine la solution un problème en résolvant des **instances plus petites** du **même problème**.

Cela se traduit dans le programme par un **indice** (image de la taille du problème) dans l'appel de la fonction qui va diminue jusqu'à **un cas de base** où la solution peut être calculée directement (sans passer par un appel récursif).

Chaque appel récursif doit se faire sur une **instance plus petite** du **même problème** : pour que le programme termine, il faut finisse par arriver sur un cas de base.

Récursivité (fonction récursive)

- l'argument de la fonction doit contenir un **indice** qui va diminuer à chaque appel : l'appel de la fonction se fait sur une **instance plus petite** du **même problème**
- l'appel récursif s'arrête sur un **cas de base** où la solution est calculée simplement

Prototype d'une fonction récursive

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
def fonction_recursive(n) :
    " prototype d'une fonction récursive"
    if n == 0 :
        # Cas de base
        # calcul de la solution
        return valeur_0
    else :
        # Cas général
        # calcul de la valeur à retourner
        # + appel recursif sur une instance plus petite
        # du même problème (n -> n-1)
        return valeur_n + fonction_recursive(n-1)
```

Exemple de la fonction factorielle

Algorithme: `factorielle_recursive (n)`
Entrée :

- `n` : un entier $n \geq 0$

Sortie :

- la valeur de $n! = n \times (n-1) \times \cdots \times 2 \times 1$

Procédure :

1. Si $n = 0$, alors retourner 1
2. Sinon, retourner $n \times$`factorielle_recursive(n-1)`