ITC – chapitre n°11
Algorithmes gloutons

I. Un exemple introductif : le rendu de monnaie

Considérons le problème suivant : comment rendre la monnaie sur une somme donnée, en minimisant le nombre de pièces/billets utilisées ? Une façon assez directe de faire est la suivante : on prend la plus grosse coupure disponible qui soit inférieure à la somme à rendre, et on recommence avec la somme restante, jusqu'à atteindre zéro. En supposant que l'on dispose d'un tableau contenant les billets/pièces existants en ordre décroissant, par exemple

```
In [1]: coupure_euros = [500, 200, 100, 50, 20, 10, 5, 2, 1]
In [2]: retour_monnaie(79, coupures_euros)
Out[1]: [50, 20, 5, 2, 2]
```

l'analyse précédente appelle une écriture récursive :

```
def retour_monnaie(somme, coupures_dispo):
    """
    Renvoie le découpage glouton du rendu de monnaie.
    On considère que chaque coupure est disponible en quantité infinie.
    Args:
        somme (int): la quantité de monnaie à rendre
        coupures_dispo  (list[int]): les coupures possibles
    Returns:
        rendu (list[int]): la liste des coupures à rendre
    Préconditions:
        coupures_dispo trié dans l'ordre croissant
    """
    if somme == 0:  # cas de base
        return []
    i_coup = 0
    # on cherche la plus grande coupure compatible
    while coupures_dispo[i_coup] > somme:
        i_coup += 1
    coup = coupures_dispo[i_coup]
    return [coup] + retour_monnaie(somme-coup, coupures_dispo)
```

On conçoit donc que cette approche fonctionne ainsi : on cherche un choix localement optimal (s'il faut rendre 33€, un billet de 50€ n'est pas acceptable, mais un billet 20€ est mieux que un de 10€) qui réduit le problème à traiter à un sous-problème (rendre les 13€ restants). On espère alors que la résolution de sous-problèmes amènera à la meilleure solution possible.

On pourra, à titre d'exercices :

- ☞ proposer une version itérative de `retour_monnaie`
- ☞ l'adapter au cas où on a une quantité limitée de chaque pièces/billets : on pourra transmettre un argument `caisse` de type dictionnaire, dont les clés sont exactement les éléments de `coupures`, et les valeurs le nombre de chaque pièce/billet disponibles ; par exemple une caisse pourra être définie par

```
caisse_départ = {1:50, 2:50, 5:50, 10:50, 20:50, 50:10, ...}
```

avant d'être passée en paramètre. Elle pourra être modifiée par la fonction `retour_monnaie`.

II. Algorithmes gloutons

2.1. Considérations générales

Définition – Algorithme glouton

Un algorithme glouton est un algorithme d'optimisation au cours duquel on choisit une solution partielle qui semble optimale localement, puis on considère le sous-problème restant et ré-applique la même méthode. Il ne remet jamais en cause une décision prise au cours des étapes précédentes.

Pour pouvoir envisager un algorithme glouton, il faut considérer les questions suivantes :

- a-t-on affaire à un problème d'optimisation ?
- est-il possible de construire une solution à partir d'un « morceau de solution » et d'un sous-problème restant ?[1]
- est-il possible d'identifier un « morceau de solution » localement optimal ?

Les algorithmes gloutons ne garantissent pas d'obtenir la meilleure solution globale pour tous les problèmes (nous verrons en TD un contre-exemple célèbre, le problème du sac à dos). Il permettent néanmoins d'obtenir en un temps de calcul raisonnable **une** solution.

[1] Ce deuxième élément explique qu'on implémente souvent les algorithmes gloutons sous forme récursive.

2.2. Retour sur le rendu de monnaie

Dans le cas du rendu de monnaie, l'algorithme glouton représente une façon naturelle de rendre la monnaie. On retrouve bien les trois éléments précédemment évoqués :

- il s'agit d'un problème d'optimisation ; on veut utiliser le moins de coupures possibles ;
- on peut construire un solution en prenant une coupure de valeur v inférieure à la somme à rendre s (« morceau de solution ») et en retrouvant le sous-problème de faire la monnaie sur $s - v$;
- on peut définir un « morceau de solution » localement optimal : la plus grande coupure de valeur $v \leqslant s$.

Notons que l'algorithme glouton donne ici la solution globalement optimale pour le système euro, mais cela dépend du système monétaire. Par exemple, prenons l'ancienne division des livres sterling anglaises (avant 1971) :

- la livre sterling était divisée en 20 shillings ;
- chaque shilling vaut 12 pence (pluriel de penny).

La valeur des différentes pièces existantes était 1, 3, 4, 6, 12, 24, 30, 60, 120 et 240 pence. Rendre la monnaie avec l'algorithme glouton sur 48 pence conduit alors à

$$1 \times 30 + 1 \times 12 + 1 \times 6$$

soit 3 pièces, alors que 2×24 donne moins de pièces.

> **i** Pour la culture, un système monétaire dans lequel l'algorithme glouton optimise le rendu de monnaie est dit canonique.

III. Le problème du choix d'activités

On considère le problème suivant : plusieurs cours peuvent prendre place dans une unique salle. Chaque cours a une heure de début et une heure de fin. Quels cours sélectionner afin de maximiser le nombre de cours ayant lieu dans la salle ? Ici, on peut construire une solution « gloutonne » de la manière suivante :

- on sélectionne le cours c_i démarrant au premier horaire et ayant l'heure de fin la plus tôt ; ainsi on laisse la plus grande plage horaire possible pour la suite ;
- on recommence en considérant que l'heure de début de l'occupation de la salle est à présent l'heure de fin du cours c_i.

Le fait de choisir l'activité qui garde la plus grande plage horaire pour placer d'autres cours est ce qui fait que chaque choix est localement optimal, donc glouton.

> **i** Ce problème est en fait très générique et s'applique à de nombreux autres contextes de partage d'une ressource unique, par exemple la projection de films dans une salle de cinéma ou l'attribution d'un équipement médical au plus de patients possibles pour des examens.
>
> Un problème voisin est celui de l'ordonnancement : ici les tâches n'ont pas de date de début figée, mais une durée d'exécution et une date limite. Il s'agit alors d'allouer la ressource pour exécuter le plus de tâches possibles avant leur deadline ; on retrouve notamment ce problème dans l'allocation du temps de calcul processeur à diverses tâches dans un ordinateur.

Pour notre implémentation, considérons que la liste des cours possibles soit donnée dans une liste de triplets $(cours, début, fin)$, triée par heure de fin, par exemple

```
cours = [('Maths', 9, 10), ('Physique', 8, 11), ('Anglais', 9, 12), ... ]
```

```
def choix_cours(cours, début, fin):
    """
    Programme un maximum de cours dans une salle unique.
    Args:
        cours (list[tuple]): la liste des cours (nom, début, fin)
        début, fin (int): horaires de début et de fin d'usage de la salle
    Returns:
        planning (list[str]): un ensemble de cours dans une même salle
    Préconditions:
        cours est trié par heure de fin
    """
    n, i = len(cours), 0
    # on trouve le premier cours qui débute après l'horaire de début
    while i < n and cours[i][1] < début:
        i += 1
    if i == n or cours[i][2] >= fin:  # cas de base
        return []
    return [cours[i]] + choix_cours(cours, cours[i][2], fin)
```

IV. Le problème de l'allocation des salles

Considérons à présent un problème un peu différent : on dispose à nouveau d'une liste de cours avec heures de début et de fin, mais cette fois-ci tous doivent être assurés. La question est à présent celle de leur attribuer à chacun une salle, mais **en utilisant le moins de salles possibles**. Comme pour le choix d'activités, il s'agit d'un problème pour lequel l'approche gloutonne donne la solution optimale.

L'idée est à présent la suivante :

- on regarde l'activité démarrant le plus tôt, et on regarde si une salle déjà en partie utilisée peut l'accueillir ;
- si oui, on la programme dans cette salle ; si non, on ouvre une nouvelle salle ;
- on va maintenant résoudre le problème restant avec le nouvel ensemble de salles et un cours en moins à placer.

On considère donc cette fois-ci que la liste de départ des cours est triée par ordre croissant d'heure de début.

```python
def allocation_salles(cours):
    """
    Alloue un minimum de salles pour des cours
    Args:
        cours (list[tuple]): la liste des cours  (nom, début, fin)
    Returns:
        planning (list[list[tuple]]): un ensemble de salles et
            leur planning de cours
    Préconditions:
        cours est trié par heure de début
    """
    if len(cours) == 0:  # cas de base
        return []
    salles = allocation_salles(cours[:-1])
    à_placer = cours[-1]
    for s in salles:   # si une salle existante peut prendre le cours
        if s[-1][2] <= à_placer[1]:
            s.append(à_placer)
            return salles
    salles.append([à_placer])  # on ouvre une nouvelle salle
    return salles
```