Algorithmes gloutons

I. Introduction

On veut écrire une fonction `vers_entier(t)` qui convertit un tableau d'entiers correspondant à l'écriture décimal d'entier en cet entier.

On veut, par exemple :

```
>>> vers_entier([2, 5, 3, 1, 4])
25314
```

On pourra faire cela à l'aide d'un parcours de tableau, en construisant le nombre correspondant aux chiffres lus. Par exemple, supposons que l'on est à l'indice 3 d'un tableau `t` où figure un `1` (que l'on a souligné) :

`[2, 5, 3, 1, 4]`

On a construit jusqu'à présent le nombre 253 en utilisant les trois premières valeurs du tableau.

Exercice 1

1. En lisant le `1` souligné, quel est le nouveau nombre que l'on obtient et, surtout, comment l'obtenir ?
2. En déduire le code de la fonction `vers_entier`.

II. Approche gloutonne

Étant donné une liste de chiffres (ou, plus précisément, une liste d'entiers compris entre 0 et 9), on peut former l'écriture décimale de différents nombres.

Par exemple, avec les chiffres `2`, `7`, `0`, `9`, `2`, `1` et `4`, on peut former `2741029`, `4290217`, etc. (on utilise tous les chiffres).

Exercice 2 Quel est le plus grand nombre que l'on peut former à l'aide de ces chiffres ?

Définition (Problème d'optimisation) Un **problème d'optimisation** consiste à déterminer, dans un ensemble, un élément qui maximise une certaine fonction.

Exemple Dans l'exemple précédent, il s'agit, parmi les permutations des chiffres donnés, d'en trouver une qui est l'écriture décimale du plus grand

nombre.

Pour répondre à la question précédente, nous n'avez normalement pas essayé toutes les $\frac{7!}{2!} = 2520$ permutations possibles pour résoudre le problème. Au contraire, vous avez probablement construit la solution chiffre par chiffre. Vous avez utilisé (sans le savoir) une *approche gloutonne*.

Définition (Algorithme glouton) Un **algorithme glouton** est un type d'algorithme qui construit progressivement une solution d'un problème d'optimisation en faisant, à chaque étape, le meilleur choix possible.

Exemple Dans l'exercice précédent, on construit la permutation de gauche à droite chiffre par chiffre en choisissant, à chaque fois, le plus grand parmi ceux qui restent.

On parle d'optimisation locale : on optimise chaque choix *localement*, sans essayer d'anticiper les étapes suivantes.

On a utilisé, pour former le plus grand nombre possible, le tri des chiffres par ordre décroissant. Cela peut se faire avec la fonction `sorted` qui renvoie un nouveau tableau contenant les valeurs du tableau passé en argument triées dans l'ordre croissant. Si on ajoute comme argument `reverse=True`, le tableau obtenu est trié par ordre décroissant. Par exemple,

```
>>> sorted([2, 5, 4, 1, 2])
[1, 2, 2, 4, 5]
>>> sorted([2, 5, 4, 1, 2], reverse=True)
[5, 4, 2, 2, 1]
```

Exercice 3 Écrire une fonction `plus_grand_nombre(t)` qui renvoie le plus grand nombre possible obtenu à l'aide du tableau `t`. On veut, par exemple,

```
>>> plus_grand_nombre([2, 0, 4, 1, 2])
42210
```

Exercice 4 (Attention, plus difficile, ne perdez pas trop de temps dessus) Écrire une fonction `plus_petit_nombre(t)` qui renvoie le plus petit nombre que l'on peut écrire à l'aide des chiffres du tableau `t`. On supposera que `t` contient au moins un chiffre non nul. Attention, tous les chiffres doivent apparaître dans l'écriture standard du nombre, sans `0` en tête. Par exemple, on doit avoir :

```
>>> plus_petit_nombre([2, 0, 4, 1, 2])
10224
```

III. Sélection d'activités

Vous avez une liste d'activités, chacune avec une date de début et de fin. Par exemple, dans la figure 1 ci-dessous, il y a 11 activités $A_1, \ldots, A_{11}$, dont par exemple l'activité A_7 qui débute au temps 4 et se termine au temps 7.

L'objectif est de sélectionner le maximum d'activités compatibles, c'est-à-dire de telle sorte qu'il n'y ait jamais deux activités simultanément.

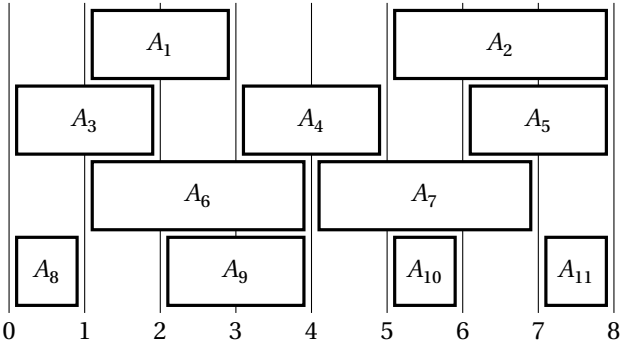

Figure 1: Exemple de liste de activités

Exercice 5 Donner, pour l'exemple précédent, une sélection de activités compatibles de cardinal maximal.

Pour obtenir une méthode générale, on se base sur une propriété d'échange :

Exercice 6 Prouver que si s est une sélection d'activités de cardinal maximal, et si a est une activité de date de fin minimale, alors on peut remplacer la première activité de s par a sans créer de problème de compatibilité.

On en déduit l'algorithme suivant :

1. On trie les activités par date de fin croissantes ;
2. On parcourt les activités en sélectionnant à chaque fois la premier activité compatibles avec celles déjà sélectionnée, c'est-à-dire la première commençant après la fin de l'activité précédente.

Exercice 7 Écrire une fonction `second(t)` qui renvoie la seconde valeur (qui a pour indice 1) du tableau `t`.
On veut, par exemple,

```
>>> second([3, "Bonjour", False])
"Bonjour"
```

On peut utiliser `second` avec le mot-clé `key` pour indiquer que le tri s'effectue selon la seconde valeur des tableaux. On peut toujours, au besoin, utiliser `reverse` :

```
>>> activites
[[1, 3], [6, 8], [0, 2], [3, 5], [5, 8]]
>>> sorted(activites)
[[0, 2], [1, 3], [3, 5], [5, 8], [6, 8]]
>>> sorted(activites, key=second)
[[0, 2], [1, 3], [3, 5], [6, 8], [5, 8]]
>>> sorted(activites, key=second, reverse=True)
[[6, 8], [5, 8], [3, 5], [1, 3], [0, 2]]
```

Exercice 8 Écrire une fonction `selection(activites)` qui renvoie une liste contenant une quantité maximales de activités compatibles.
On rappelle l'algorithme : on parcourt les activités par date de fin croissantes, et on sélectionne à chaque fois une activité si elle débute après la fin de la précédente.
On doit avoir, par exemple :

```
>>> activites = [[1, 3], [6, 8], [0, 2], [3, 5], [5, 8]]
>>> selection(activites)
[[0, 2], [3, 5], [6, 8]]
```

IV. Rendu de monnaie

Nous allons maintenant nous intéresser à un autre problème, celui du rendu de monnaie : on a une somme n à rendre en monnaie, et la question est de déterminer, étant donné une liste de valeurs de pièces ou billets, comment le faire avec le moins de pièces possibles (encore un problème d'optimisation).

L'approche gloutonne est la suivante : à chaque étape, on utilise une pièce ou une billet de plus grand valeur possible.

Par exemple, avec des pièces de 1, 2, 5 et 10 centimes, pour rendre 9 centimes en monnaie, on commence par donner une pièce de 5 centimes.

Exercice 9 Avec la même liste de pièces, quelles pièces sont utilisées pour rendre 17 centimes ?

Exercice 10 Écrire une fonction `rendu(pieces, valeur)` qui, étant donné une liste de pièces et une valeur v, renvoie une liste des pièces utilisées pour obtenir la valeur v. On supposera que :

- ⋆ la valeur v est un entier positif ou nul ;
- ⋆ la liste des pièces est classée par ordre décroissante ;
- ⋆ la plus petite valeur de pièce est 1.

On veut par exemple :

```
>>> rendu([10, 5, 2, 1], 13)
[10, 2, 1]
>>> rendu([10, 5, 2, 1], 9)
[5, 2, 2]
>>> rendu([4, 3, 1], 11)
[4, 4, 3]
```

Il existe des cas où l'approche gloutonne ne donne pas toujours le nombre minimal de pièces. Cependant, on peut montrer[1] que le système *euro*, basé sur les valeurs 1, 2, 5, 10, 20, 50, 100, etc. a la propriété que l'algorithme glouton donnent toujours le plus petit nombre de pièces et billets possibles. On dit qu'il est *canonique*. L'ancien système monétaire anglais n'était pas canonique.

Exercice 11 Montrer que le système monétaire constitué de pièces de valeurs 1, 3 et 4 n'est pas canonique, c'est-à-dire qu'il existe une valeur n pour laquelle l'approche gloutonne ne donne pas le nombre minimal de pièces.

Cela illustre que, parfois, une approche gloutonne n'est pas optimale, et qu'il faut prouver en général prouver son optimalité. Dans les situations où un algorithme glouton n'est pas optimal, il a néanmoins l'avantage de la simplicité et de la rapidité, et permet d'obtenir facilement une solution exploitable à défaut d'être optimale (et il existe de nombreuses situations où l'on ne sait pas obtenir de solution optimale dans des temps raisonnables).

V. Allocation de salles

Nous allons pour finir étudier un nouveau problème, proche de celui de sélection d'activité : étant donné une liste d'activités, quel est le nombre minimal de salles nécessaires pour que toutes les activités puissent avoir lieu, chaque salle abritant au plus une activité à tout moment.

Reprenons l'exemple précédent, en parcourant la liste des activités par dates de fin croissantes. Dans une première salle commence l'activité `[0, 2]`. On a besoin d'une second pièce pour la seconde activité `[1, 3]` qui débute avant la fin de l'autre. Il est clair que deux salles suffisent :

- ⋆ Salle 1 : `[0, 2]`, `[3, 5]`, `[5, 8]`

[1] Mais nous l'admettrons ici.

⋆ Salle 2 : `[1, 3]`, `[6, 8]`

Exercice 12 Déterminer le nombre minimale de salles nécessaires pour la liste des activités de la figure 1.

Une façon de résoudre ce problème est le suivant : on trie les différentes activités par date de début (et non plus de fin), et on les parcourt par début croissant. À chaque activité, si possible, on l'ajoute à une salle libre (à cet horaire). Si ce n'est pas possible, on créée une nouvelle salle pour cette activité.

Pour programmer cela, il suffit d'avoir une liste de longueur le nombre de salles, et contenant, pour chaque salle, l'horaire de fin de la dernière activité attribuée à cette salle. Avec la liste `[[1, 3], [6, 8], [0, 4], [2, 5], [5, 8]]`, après classement par ordre de début croissant, on a :

```
[[0, 4], [1, 3], [2, 5], [5, 8], [6, 8]]
```

On a alors le comportement suivant, où on souligne à chaque fois la nouvelle date de fin :

Activité	Fin pour chaque salle
`[0, 4]`	`[4]`
`[1, 3]`	`[4, 3]`
`[2, 5]`	`[4, 3, 5]`
`[5, 8]`	`[8, 3, 5]`
`[6, 8]`	`[8, 8, 5]`

Exercice 13 Écrire une fonction `allocation` qui implémente cet algorithme : elle prends en entrée une liste d'activités représentées par un couple *date de début* / *date de fin* et qui renvoie le nombre minimal de salles nécessaires pour les organisée.

Exercice 14 Nous allons maintenant prouver que cet algorithme est correct.

1. Montrer que le nombre de salles indiqué par la fonction `allocation` est suffisant pour que chaque activité puisse avoir lieu seule dans sa salle.

2. Montrer qu'il n'est pas possible de le faire avec moins de salles.