Algorithmes gloutons

Objectifs du TP

Avant la séance : faire le travail préparatoire pour comprendre les algorithmes gloutons en les déroulant à la main, avec une feuille de papier et sans ordinateur.

Priorité pendant la séance : coder deux algorithmes gloutons (rendu de monnaie et sélection d'activités) **Q.4** et **Q.8**.

Une fois que les deux algorithmes sont codés : répondre aux autres questions sur les deux algorithmes précédents et coder un dernier algorithme glouton (problème du sac à dos).

0 Travail préparatoire

Ce travail préparatoire est **à faire obligatoirement avant de venir en TP d'informatique.** Vous n'avez rien à programmer, il faut dérouler les algorithmes à la main, avec un crayon et une feuille de papier.

0.1 Problème du rendu de monnaie

Lorsqu'on achète un objet en espèces, le commerçant doit nous rendre la monnaie. Par exemple, si l'on doit 11 € au commerçant et qu'on lui donne un billet de 20 €, il doit nous rendre 9 €.

Le problème du rendu de monnaie consiste à déterminer la manière de rendre la monnaie avec un minimum de pièces et de billets. Ici, parmi toutes les combinaisons possibles (9 pièces de 1 €, 4 pièces de 2 € et une de 1 €, etc.) la manière optimale de rendre 9 € est d'utiliser un billet de 5 € et deux pièces de 2 €.

Dans toute la suite, on considérera le système de pièces et de billets en euros, sans tenir compte des centimes. Pour simplifier, on appellera « pièces » les pièces et les billets.

On travaille avec le système de pièces en euros :

1 €, 2 €, 5 €, 10 €, 20 €, 50 €, 100 €, 200 € et 500 €.

Q.1 Dénombrer (à la main, avec un crayon et une feuille de papier...) toutes les combinaisons permettant de rendre 9 €.

Le problème du rendu de monnaie est un problème d'optimisation combinatoire : on cherche à optimiser une grandeur (le nombre de pièces à rendre), mais il y a beaucoup de combinaisons possibles. On va s'intéresser à une classe d'algorithmes qui permettent de déterminer une solution optimale ou proche de l'optimum sans avoir besoin d'explorer toutes les combinaisons : les algorithmes gloutons.

L'algorithme glouton pour le rendu de monnaie consiste à considérer le montant qu'on doit rendre, à sélectionner à chaque fois la plus grande pièce possible et à faire ceci tant que le montant à rendre est strictement positif. Par exemple, si on a 26 € à rendre, on commence par rendre un billet de 20 €, puis un billet de 5 €, puis une pièce de 1 €. Pour rendre 42 €, on rend d'abord un premier billet de 20 €, puis un deuxième, puis une pièce de 2 €.

L'idée générale des algorithmes gloutons est de chercher à faire une optimisation locale : à une étape donnée, on fait le choix qui minimise le nombre de pièces à rendre, mais rien ne dit que cela donne une optimisation globale (on verra un exemple dans le TP).

Une solution non optimale donnée par l'algorithme glouton est appelée **heuristique gloutonne.** Une heuristique gloutonne est en général une très bonne valeur approchée de la vraie solution optimale et elle est obtenue avec un temps de calcul beaucoup plus faible.

Q.2 Déterminer, en appliquant l'algorithme glouton à la main, les pièces à rendre pour les montants suivants : 13 €, 47 €, 187 €.

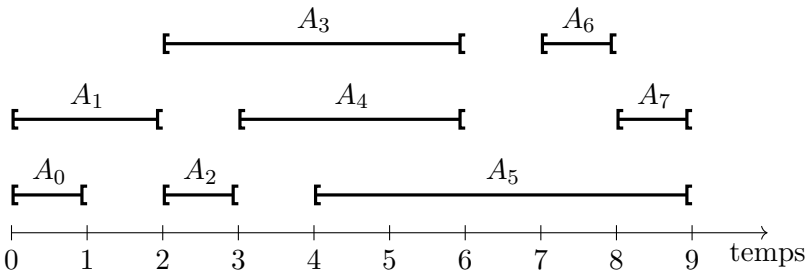
0.2 Problème de la sélection d'activités

Dans le problème de la sélection d'activités, on considère un ensemble d'activités définies par un horaire de début et un horaire de fin. On cherche à déterminer le nombre maximal d'activités qu'on peut sélectionner sans que leurs horaires se chevauchent.

Les activités en question peuvent être les cours suivis par un étudiant de CPGE, qui ne peut pas être en même temps en TP d'informatique et en cours de maths.

On se donne un ensemble d'activités A_i caractérisées par un couple (d_i, f_i) correspondant à une heure de début d_i et une heure de fin f_i. Une activité se déroule dans l'intervalle de temps $[d_i, f_i[$. Deux activités A_i et A_j sont donc en conflit lorsque $[d_i, f_i[\bigcap[d_j, f_j[\neq \emptyset$.

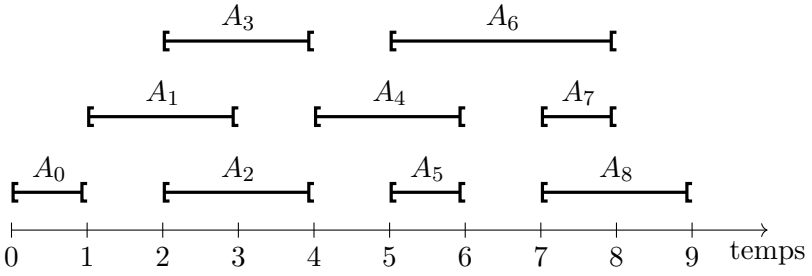
Pour simplifier, on prendra des valeurs entières de d_i et f_i. Sur le diagramme ci-dessous, on a par exemple $d_0 = 0$, $f_0 = 1$, $d_1 = 0$, $f_1 = 2$, etc.



L'algorithme glouton pour la sélection d'activités consiste à trier les activités par heure de fin croissante, à prendre comme première activité celle qui finit le plus tôt. Pour choisir l'activité suivante, on parcourt la liste des activités restantes par heure de fin croissante et on choisit la première compatible avec l'activité précédente (c'est donc l'activité compatible qui finit le plus tôt), et ainsi de suite.

Dans l'exemple précédent, la suite d'activités donnée par l'algorithme glouton est A_0, A_2, A_4, A_6, A_7.

Q.3 Déterminer à la main les deux suites d'activités que peut donner l'algorithme glouton pour les activités du diagramme ci-dessous. (Il y a deux suites possibles, toutes les deux optimales au sens choisi : on maximise le nombre d'activités, pas leur durée totale).



1 Rendu de monnaie

1.1 Algorithme glouton et ses limites

On veut écrire une fonction déterminant les pièces à rendre avec un algorithme glouton pour un montant donné dans un système de pièces donné. Le système de pièces sera codé sous la forme d'une liste de valeurs croissantes. Pour le système de pièces en euros (sans les centimes), on pourra écrire par exemple.

```
systeme_euro = [1, 2, 5, 10, 20, 50, 100, 200, 500]
```

Q.4 Écrire une fonction `rendu_monnaie(montant, systeme)` prenant en arguments :

- `montant`, un entier égal au montant qu'on veut rendre
- `systeme`, une liste associée à un système de monnaie

et renvoyant une liste contenant les valeurs des pièces à rendre. Comportement attendu :

```
>>> rendu_monnaie(9, systeme_euro)
[5, 2, 2]
>>> rendu_monnaie(100, systeme_euro)
[100]
>>> rendu_monnaie(82, systeme_euro)
[50, 20, 10, 2]
>>> rendu_monnaie(0, systeme_euro)
[]
```

Pour le système de pièces en euros, on peut montrer que la solution fournie par l'algorithme glouton est bien la solution optimale. On dit que ce système de pièces est canonique (c'est le cas de presque tous (voire tous ?) les systèmes en usage actuellement dans le monde).

Avant 1971 [1], la livre anglaise (symbole £) était divisée en 240 *pence* (singulier *penny*). Les pièces en circulation étaient les suivantes (on s'épargne les billets...) :

Dénomination	valeur en *pence*
Farthing	1/4 *penny*
Half Penny	1/2 *penny*
Penny	1 *penny*
Threepence	3 *pence*
Sixpence	6 *pence*
Shilling	12 *pence*
Florin	24 *pence*
Half Crown	30 *pence*
Crown	60 *pence*

Si l'on oublie le *farthing* et le *halfpenny*, le système peut être codé de la manière suivante

```
systeme_UK = [1, 3, 6, 12, 24, 30, 60]
```

Q.5 *(Pour les plus rapides.)* Que renvoie `rendu_monnaie(48, systeme_UK)` ? Quelle est la solution optimale ?

Un algorithme glouton ne renvoie pas toujours une solution optimale, mais la recherche d'une solution optimale par un autre algorithme peut demander un temps d'exécution beaucoup plus long (ou un effort de programmation beaucoup plus important).

1. Depuis, la livre a été décimalisée : il y a 100 pennys dans une livre.

1.2 Notion de variant de boucle

Un variant de boucle est une expression dont la valeur varie à chacune des itérations de la boucle. Un variant bien choisi permet de montrer qu'une boucle Tant Que (`while`) termine.

Considérons par exemple le programme suivant qui affiche les entiers par ordre décroissant de 10 à 1.

```
i = 10
while i > 0:
    print(i)
    i = i-1
```

Un bon variant de boucle est ici `i` : en effet, `i` est entier, strictement décroissant à chaque itération de la boucle. Or par la condition de la boucle, on a `i > 0`. On en conclut que la boucle `while` termine.

Le variant de boucle peut avoir une expression quelconque. Considérons la boucle ci-dessous :

```
i = 1
while i < 10:
    print(i)
    i = i+1
```

On peut prendre comme variant de boucle la grandeur `9-i` : il s'agit d'un entier strictement décroissant à chaque itération de la boucle. Or la condition de la boucle est `i < 10`, on a nécessairement `9-i >= 0`. Puisque `9-i` est strictement décroissant, on en conclut que la boucle termine.

Sur ces deux exemples élémentaires, la notion de variant n'apporte pas grand-chose : il est évident que les boucles terminent. Pour des programmes plus complexes, la notion de variant de boucle est très utile. Elle permet d'éviter les boucles infinies.

Q.6 *(Pour les plus rapides.)* Identifiez un variant de boucle pour l'algorithme glouton et montrez qu'il termine. (Il peut être pratique de modifier la condition de la boucle `while` pour faciliter le raisonnement).

1.3 Rendu de monnaie avec des centimes et représentation des nombres en machine

Il est tentant de généraliser le rendu de monnaie avec la totalité des pièces en euros, y compris les centimes, en définissant le système suivant :

```
systeme_euro_complet = [0.01, 0.02, 0.05, 0.10, 0.20, 0.50,
                                    1, 2, 5, 10, 20, 50, 100, 200, 500]
```

Pour rendre un euro cinquante, tout va bien :

```
>>> rendu_monnaie(1.50, systeme_euro_complet)
[1, 0.5]
```

Mais pour certaines valeurs, le code ne fonctionne plus. Les valeurs en question dépendent de la manière dont vous avez écrit votre code, mais vous pouvez tester les instructions suivantes, pour rendre un euro cinquante-et-un, ou bien un euro soixante-treize.

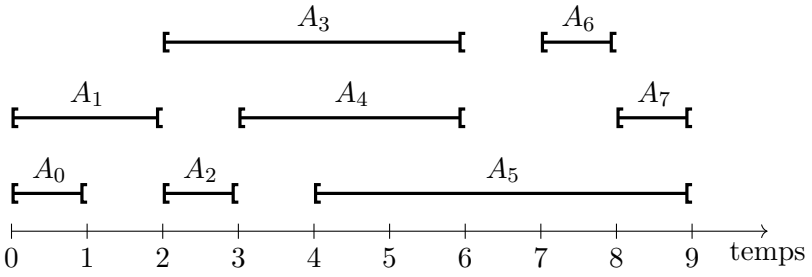
```
>>> rendu_monnaie(1.51, systeme_euro_complet)
>>> rendu_monnaie(1.73, systeme_euro_complet)
```

Selon le code que vous avez choisi, au moins une de ces instructions devrait poser problème. Ce problème est lié à la représentation des nombres en machine.

Q.7 *(Pour les plus rapides.)* Identifiez la cause exacte du problème, proposez une solution et codez-la.

2 Sélection d'activités

On va représenter les activités par une liste de listes `activites`. Par exemple, les activités suivantes



sont représentées par la liste ci-dessous :

```
activites = [[0, 0, 1], [1, 0, 2], [2, 2, 3], [3, 2, 6],
             [4, 3, 6], [5, 4, 9], [6, 7, 8], [7, 8, 9]]
```

Chaque élément de la liste `activites` est une liste à trois éléments correspondant successivement au numéro de l'activité, à son heure de début et à son heure de fin. La sous-liste `[3, 2, 6]` représente ainsi l'activité A_3, qui commence à l'instant $d_3 = 2$ et finit à l'instant $f_3 = 6$.

On donne ci-dessous le code de la fonction `trie_activites(activites)` qui prend en argument une liste `activites` et renvoie une nouvelle liste contenant les mêmes activités triées par heure de fin croissante. Pour effectuer le tri, on utilise une fonction annexe `heure_de_fin(activite)` qui renvoie l'heure de fin d'une activité. Vous pouvez copier ces deux fonctions dans votre code.

```
1  def heure_de_fin(activite):
2      """Fonction annexe utilisée dans trie_activites qui renvoie
3      l'heure de fin d'une activité (son élément d'indice 2)"""
4      return activite[2]
5
6  def trie_activites(activites):
7      """Trie une liste d'activités par temps de fin croissant"""
8      return sorted(activites, key=heure_de_fin)
```

On propose ci-dessous un pseudo-code pour la sélection d'activités :

Algorithme 1 : Algorithme glouton pour la sélection d'activités

Entrées : *activites* : une liste d'activités non triées
Sorties : *activites_selectionnees* : une liste d'activités sélectionnées
Fonction *selectionne_activites(activites)*
  activites_triees ← la liste des activités triées par heure de fin croissante ;
  activites_selectionnees ← la liste vide ;
  ajouter l'activité finissant le plus tôt à *activites_selectionnees* ;
  pour chaque *activite* ∈ *activites_triees* **faire**
    si l'heure de début de *activite* est compatible avec l'heure de fin de la dernière activité sélectionnée **alors**
      ajouter *activite* à *activites_selectionnees* ;
    fin
  fin
  retourner *activites_selectionnees*
fin

Q.8 Écrivez une fonction `selectionne_activites(activites)` prenant en argument une liste d'activités et retournant une liste d'activités sélectionnées par l'algorithme glouton. Comportement attendu si `activites` est la liste donnée en exemple plus haut :

```
>>> print(selectionne_activites(activites))
[[0, 0, 1], [2, 2, 3], [4, 3, 6], [6, 7, 8], [7, 8, 9]]
```

On peut montrer que l'algorithme glouton pour la sélection d'activités est optimal, mais cela sort du cadre de ce TP.

3 Problème du sac à dos

3.1 Une première version du problème du sac à dos

Dans le problème du sac à dos, on veut remplir un sac à dos de capacité connue (en kilogrammes) avec des objets, de manière à maximiser la valeur totale des objets contenus dans le sac à dos, sans dépasser sa capacité. On dispose pour cela d'une liste de listes `objets` contenant des objets caractérisés par leur nom, leur masse et leur valeur.

Par exemple, la liste ci-dessous correspond à un objet A de masse 1 kg et de valeur 10 €, un objet B, de masse 2 kg et de valeur 5 €, etc.

```
objets = [['A', 1, 10], ['B', 2, 5], ['C', 5, 5], ['D', 2, 8],  ['E', 3, 6]]
```

Pour remplir un sac à dos de capacité 6 kg, le choix des objets A, B et D pour une valeur totale de 23 € et d'une masse totale de 5 kg est préférable au choix des objets A et C, dont la valeur totale est de 15 € pour une masse totale de 6 kg.

L'algorithme glouton pour le problème du sac à dos consiste à trier les objets par valeur massique décroissante (c'est-à-dire par prix au kilo décroissant), et à remplir progressivement le sac avec les objets de plus grande valeur massique tant que la capacité du sac n'est pas dépassée.

Q.9 Écrivez une fonction `trie_objets(objets)` prenant comme argument une liste d'objets et renvoyant la liste de ces objets triés par valeur massique décroissante. Vous pourrez pour cela utiliser la fonction `sorted` dont la documentation est disponible en ligne : https://docs.python.org/fr/3/library/functions.html#sorted Résultat attendu avec la liste précédente :

```
>>> trie_objets(objets)
[['A', 1, 10], ['D', 2, 8], ['B', 2, 5], ['E', 3, 6], ['C', 5, 5]]
```

En appliquant l'algorithme glouton à la liste précédente, on commence par prendre l'objet A, puis on prend les objets D et B, jusqu'à un total de 5 kg. On ne peut plus ensuite prendre les objets E et C.

Q.10 Écrivez une fonction `sac_a_dos(capacite, objets)` prenant en arguments la capacité du sac à dos et une liste d'objets et renvoyant la liste des objets sélectionnés par l'algorithme glouton ainsi que leur valeur totale. Résultat attendu avec la liste précédente :

```
>>> sac_a_dos(6, objets)
([['A', 1, 10], ['D', 2, 8], ['B', 2, 5]], 23)
```

Q.11 L'algorithme glouton pour le problème du sac à dos ne donne pas toujours la solution optimale. Proposez une liste d'objets pour laquelle l'algorithme glouton ne donne pas le chargement de plus grande valeur.

3.2 Problème du sac à dos fractionnaire

Une variante du sac à dos consiste à considérer que les objets sont sécables, et qu'on peut choisir une fraction quelconque de chaque objet. Dans ce cas, l'algorithme glouton consiste à trier les objets par valeur massique décroissante, puis à remplir le sac à dos avec la plus grande fraction possible de l'objet de plus grande valeur massique, avant de passer au suivant, etc.

Pour la liste d'objets donnée en exemple, on sélectionne la totalité des objets A, D et B, pour un total de 5 kg, puis on complète le sac avec 1 kg d'objet E, ce qui ajoute 2 € à la valeur contenue dans le sac à dos pour un total de 25 €.

Q.12 Écrivez une fonction `sac_a_dos_fractionnaire(capacite, objets)` prenant en arguments la capacité du sac à dos et une liste d'objets et renvoyant la liste des objets sélectionnés par l'algorithme glouton dans le cas fractionnaire ainsi que leur valeur totale. Résultat attendu avec la liste précédente :

```
>>> sac_a_dos_fractionnaire(6, objets)
([['A', 1, 10], ['D', 2, 8], ['B', 2, 5], ['E', 1, 2.0]], 25.0)
```

On peut montrer que dans le cas du problème du sac à dos fractionnaire, l'algorithme glouton donne une solution optimale.