Chapitre :
Algorithmique

Définition Un **algorithme** est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre une classe de problèmes
Le mot algorithme vient du nom d'un mathématicien perse du IXe siècle, Al-Khwârizmî. Une autre étymologie dit qu'un algorithme est un calcul (« arithmos » en grec), qui est tellement long et difficile à faire à la main qu'il en devient douloureux : « algos » signifie douleur en grec : un algorithme est un calcul pénible à faire à la main. Le domaine qui étudie les algorithmes est appelé l'algorithmique.

I. Validité d'un algorithme

Lorsqu'on écrit un algorithme, il est impératif de vérifier que cet algorithme va produire un résultat en un temps fini et que ce résultat sera correct dans le sens où il sera conforme à une spécification précise. Nous dirons que l'algorithme est valide.
On dit qu'un algorithme est **correct** si, quelque soit l'instance du problème (les valeurs des paramètres donnés en entrée) il se termine en produisant la bonne sortie.
Lorsqu'un algorithme est construit avec un boucle, pour prouver qu'il est correct on utilise la notion d'**invariant de boucle**.

Définition Un invariant de boucle est une propriété vérifiée avant l'entrée dans une boucle, à chaque passage dans la boucle, et par conséquent à la sortie de la boucle.

Méthode Pour démontrer qu'une propriété est un invariant de boucle, on commence donc par vérifier que la propriété est vraie avant l'entrée de la boucle. On appelle cette étape l'**initialisation**.
On démontre ensuite que si la propriété est vraie en entrée de boucle, alors elle est vraie après une itération. On appelle cette étape l'**hérédité**.
Ces deux étapes étant effectuées, on peut en déduire que la propriété est vraie à la sortie de la boucle, donc qu'elle est bien un invariant de boucle.

Exemple Voici un algorithme dont on souhaite démontrer qu'il permet de calculer le produit de deux entiers naturels a et b :

```
Entrée :
    a et b, deux entiers naturels
Sortie :
    p, un entier naturel
Traitement :
    m prend la valeur 0
    p prend la valeur 0
    Tant que m<a Faire
    |   m prend la valeur m+1
    |   p prend la valeur p+b
    Fin Tant que
```

La propriété « p = m×b » est un invariant de la boucle Tant que.

- **Initialisation :** Juste avant la boucle m=p=0, donc m×b=0, et donc p=m.

- **Hérédité :** On suppose que p=m×b avant une itération de la boucle.
  On note p' et m' les valeurs des variables p et m après l'itération de la boucle.
  On doit donc montrer que p'=m'×b.
  Or m'=m+1 et p'=p+b, donc m'×b=(m+1)b=mb+b=p+b=p'

On vient donc de démontrer que la propriété « p = m×b » est un invariant de la boucle.
On en déduit qu'elle est vraie à la sortie de la boucle. Or, à la sortie, on a m = a.
Donc p = a×b, ce qu'il fallait démontrer.

En plus de la correction, il faut démontrer la terminaison pour que l'algorithme soit valide. Autrement dit il faut démontrer que l'algorithme finit toujours par donner un résultat. Quand il y a des boucles, cela revient à dire que l'on sort forcément de la boucle au bout d'un nombre fini d'itérations.

Méthode Dans le cas d'une boucle non conditionnelle (boucle Pour), il est généralement certain que la boucle se termine (on sait à l'avance le nombre d'itération de la boucle si les valeurs prises par le compteur ne sont par influencées par les instructions de la boucle).
Dans le cas d'une boucle conditionnelle (boucle Tant que), ce n'est pas une évidence, et dans ce cas on utilise la notion de **variant**.

Définition Un variant est une expression (la plus simple possible étant une variable) dont la valeur converge en un nombre fini d'étapes vers une valeur pour laquelle la condition d'arrêt de la boucle est satisfaite.

Exemple Voici un algorithme court avec une boucle Tant que :

```
Entrée :
    a, un nombre entier
Sortie :
    x, un nombre entier
Traitement :
    x prend la valeur 0
    Tant que x×2<a Faire
    |  x prend la valeur x+1
    Fin Tant que
```

On peut prendre comme variant la variable x. Elle augmente de 1 à chaque itération, donc il y aura nécessairement une étape où x×2 sera supérieur à a (a étant constante).

Exemple Pour démontrer que l'algorithme qui fait le produit de a par b est valide, il reste à démontrer que l'algorithme termine. C'est le cas puisque la variable m est un variant. Elle augmente toujours de 1, donc elle finira par atteindre a (il y a en fait exactement m passages dans la boucle).

 Ne pas confondre **variant de boucle** et **invariant de boucle**. Le premier permet de démontrer la terminaison, alors que l'autre permet de démontrer la correction.

II. Coût d'un algorithme (en temps)

Le **coût**, ou la **complexité** d'un algorithme est le nombre d'opérations nécessaires à l'obtention du résultat, en fonction des données en entrée.
Le nombre d'opérations étant lié au temps passé à les réaliser, cela donne donc une idée du temps demandé par l'algorithme en fonction de la taille des données (par exemple la longueur d'une liste, ou même le nombre de chiffres dans un nombre en entrée).

Cette définition très large cache beaucoup de nuances que l'on ne détaillera pas ici. Le plus souvent, on peut s'interroger sur la complexité dans le pire cas (c'est à dire dans le cas où les données demandent le temps d'exécution le plus long).

Exemple Reprenons l'exemple de l'algorithme qui calcule le produit de deux entiers a et b vu précédemment.
Nous avons vu qu'il y a un nombre a de passages dans la boucle. Dans la boucle, il y a deux opérations d'affectation, où l'on fait à chaque fois une addition. Cela fait donc a×4 opérations.
Le coût est dit **linéaire**, car $4a$ est une expression linéaire de a.

Plus généralement :

Définition Une complexité est **linéaire** si, n étant la taille des données, elle s'exprime sous la forme $\alpha n + \beta$ avec $\alpha \neq 0$.

Définition Une complexité est **quadratique** si, n étant la taille des données, elle s'exprime sous la forme $\alpha n^2 + \beta n + \gamma$, avec $\alpha \neq 0$.

On trouve des complexités quadratiques dans le cas des boucles imbriquées. Par exemple de la forme :

```
Pour i allant de 1 à n Faire
|  Pour j allant de 1 à n Faire
|  |  ...
|  Fin Pour
Fin Pour
```

ou même

```
Pour i allant de 1 à n Faire
|  Pour j allant de 1 à i Faire
|  |  ...
|  Fin Pour
Fin Pour
```

Mais cela peut rester linéaire comme dans le cas ci-dessous, si k est une constante, ne dépendant pas de i :

```
Pour i allant de 1 à n Faire
|  Pour j allant de 1 à k Faire
|  |  ...
|  Fin Pour
Fin Pour
```

Remarque On peut s'intéresser à d'autres types de complexités (ou coûts), comme par exemple en mémoire plutôt qu'en temps.

III. Tableaux

Nous étudions ici quelques algorithmes portant sur des tableaux. Nous écrirons ces algorithmes dans le langage Python, et les tableaux seront des éléments de type list.

1. Calcul de moyenne

Pour le calcul d'une moyenne, on utilise un accumulateur, c'est à dire une variable qui va être augmenté successivement des valeurs de la liste en entrée.

```
def moyenne(liste):
    n = len(liste)
    s = 0
    for u in liste:
        s = s+u
    return s/n
```

La variable `u` prend successivement les valeurs de la liste `liste`. On dit que l'on fait un parcourt séquentiel de la liste.
Le coût de cet algorithme est linéaire en la taille `n` de la liste. En effet, il y a `n` itérations de la boucle, dans laquelle on fait deux opération (somme et affectation).
L'instruction `len(liste)`, qui permet d'obtenir la longueur de la liste, s'effectue à coût constant.

2. Recherche d'occurrence

Soit `t` un tableau et `x` un élément. On cherche l'indice `i`, s'il existe, tel que `t[i]` = x.

```
def recherche(x,t):
    n = len(t)
    i = 0
    while i<n and x != t[i]:
        i = i+1
    if i < n:
        return i
```

Déterminons le coût. Le pire des cas est celui où l'élément n'est pas dans `t`, ou si c'est le dernier élément. Dans ce cas on parcourt tout le tableau, et on produit donc `n` comparaisons. Le coût est donc linéaire.
On peut également utiliser l'algorithme suivant, qui utilise une boucle Pour de laquelle on sort prématurément avec un `return` si l'élément est trouvé :

```
def recherche(x,t):
    for i in range(len(t)):
        if t[i] == x:
            return i
```

Remarque Lorsque l'on souhaite accéder à la fois à l'élément `e` d'une liste et à son indice `i`, on peut utiliser la fonction `enumerate` :

```
def recherche(x,t):
    for i,e in enumerate(t):
        if e == x:
            return i
```

3. Recherche d'un extremum

On recherche, dans une liste, un maximum et/ou un minimum. Pour cela on va utiliser une variable qui va prendre pour valeur un maximum (ou minimum) provisoire, en commençant par le premier élément de la liste (supposée non vide).

```python
def maximum(liste):
    maxi = liste[0]
    for x in liste:
        if x > maxi:
            maxi = x
    return maxi
```

Là aussi, le coût est linéaire en la taille n de la liste.
Comme exercice, on peut modifier l'algorithme pour qu'il détermine le minimum, voire qu'il détermine à la fois le minimum et le maximum (en renvoyant alors un couple de valeurs).

4. Recherche dichotomique

Le principe de la dichotomie est de diviser par deux, à chaque itération, la taille de la liste dans laquelle on recherche un élément. Ce principe permet alors d'obtenir des algorithmes meilleurs que des algorithmes ayant un coût linéaire. On dit que la dichotomie utilise le principe *d*iviser pour régner.
Ici on suppose que la liste est triée. Il y a plusieurs moyens de trier une liste, en particulier avec Python, qui donne la fonction `sorted`.

```python
def recherche_par_dichotomie(x,liste):
    gauche = 0
    droite = len(liste)
    while droite - gauche > 1:
        centre = (gauche + droite) // 2
        if x < liste[centre]:
            droite = centre
        else:
            gauche = centre
    if x == liste[gauche]:
        return gauche
```

Démontrons la terminaison de l'algorithme : Le variant de la boucle est `droite - gauche`. À chaque itération, soit droite, soit gauche, prend la valeur du centre de l'intervalle [droite ; gauche]. Ainsi, l'écart diminue de moitié à chaque itération. Si la longueur de la liste n est inférieure à 2^k, alors au bout de k étapes (au plus), on a divisé la longueur de l'intervalle par 2^k, donc sa longueur est devenue inférieure ou égale à 1. Autrement dit `droite - gauche` $\leqslant 1$, ce qui est la condition de sortie de la boucle Tant que.

Remarque Pour une liste de longueur 100, comme $100 < 2^7$, il suffit de 7 étapes. Pour une liste de longueur 1000, comme $1000 < 2^{10}$, il suffit de 10 étapes.
Le coût est donc meilleur que linéaire
D'autre part, puisque, lorsque l'on cherche l'écriture en binaire d'un entier n, on fait successivement des divisions par 2 en même quantité que le plus petit k tel que $n < 2^k$, on en déduit que le coût correspond au nombre de bits de l'écriture en binaire de n.
Le nombre de fois que l'on peut diviser n par 2 est noté $\log_2(n)$.
Le coût de l'algorithme de dichotomie est donc en $\log_2(n)$ que l'on écrit souvent seulement $\log(n)$ par abus de notation.
Le log est un logarithme (fonction mathématique vue en terminale).

Démontrons maintenant la correction de l'algorithme.

Les variables gauche et droite sont initialisées respectivement à 0 et len(liste). Si x est strictement inférieur à liste[0] ou strictement supérieure à liste[droite-1], alors il n'appartient par à liste, et l'algorithme renvoie None, ce qui est correct.
On suppose maintenant que x est situé entre liste[0] et liste[droite-1]. Ajoutons un élément à liste, strictement supérieur à son plus grand élément ; par exemple liste[droite-1]+1.
Ainsi, on a la propriété suivante en entrant dans la boucle : « liste[gauche]⩽x<liste[droite] ». On va démontrer que cette propriété est héréditaire, afin de démontrer que c'est un invariant de la boucle Tant que. On suppose donc qu'elle est vraie à une étape donnée, et on doit démontrer qu'elle reste vraie après une itération de la boucle.
La variable centre est située entre gauche et droite d'après sa définition, et comme la liste est ordonnée, on peut donc affirmer que liste[gauche]⩽liste[centre]<liste[droite].

- Si x<liste[centre], alors nécessairement liste[gauche]⩽x<liste[centre]. Mais dans ce cas, droite prend la valeur centre, donc liste[gauche]⩽x<liste[droite]
- Sinon liste[centre]⩽x<liste[droite]. Mais dans ce cas, gauche prend la valeur centre, donc liste[gauche]⩽x<liste[droite].

La propriété est donc bien héréditaire.
On déduit des résultats précédents que la propriété est un invariant de boucle, et qu'elle est donc vraie à la sortie de la boucle.
Or, à la fin de la boucle, on a nécessairement droite - gauche $\leqslant 1$.
Plus précisément, droite - gauche $= 1$ car la différence est strictement positive (elle ne peut pas être nulle, car au dernier passage dans la boucle, droite - gauche $\geqslant 2$, et centre est alors strictement compris entre droite et gauche ; comme elle remplace l'une des deux, la différence reste strictement positive)
Ainsi, la propriété devient, en sortie de boucle : liste[gauche]⩽x<liste[gauche+1].
Par suite, soit x vaut liste[gauche], soit x n'appartient pas à la liste.
L'algorithme est bien correct.

Remarque En mathématiques, on peut voir la recherche d'une solution de l'équation $f(x) = 0$ sur un intervalle $[a; b]$ à l'aide d'un algorithme de dichotomie.

IV. Tri de listes

Le tri est une opération courante, et donc importante, en informatique. Trier une liste c'est ordonner ses éléments selon un ordre donné (numérique, alphabétique, lexicographique, etc.). Il existe de très nombreux algorithmes de tri, plus ou moins efficace, et surtout plus ou moins efficaces selon les propriétés initiales de la liste à trier.
Un bon exercice peut être de prendre un jeu de carte (ou seulement une des quatre couleurs de ce jeu), de le mélanger, puis d'observer notre manière de le trier. Mieux que de l'observer, de décrire pour quelqu'un d'autre une méthode permettant de trier le jeu. Il est possible d'observer déjà, pour un tri « naturel », plusieurs stratégies possibles, parfois même des mélanges de stratégies.
En première NSI, deux algorithmes sont à étudier, à savoir le tri par sélection et le tri par insertion. Ils ne sont pas forcément les plus efficaces ou les plus rapides, mais ils sont assez simples à comprendre. De meilleurs tris pourront être vus en terminale.
Précisons avant de les décrire ce que nous entendons ici par liste : une liste est un ensemble d'éléments accessibles via un indice (allant de 0 à $n-1$ s'il y a n éléments) dont on peut changer les éléments (et en particulier échanger deux éléments de celle-ci). Les éléments de cette liste sont tous de même type, et on dispose d'une relation d'ordre qui permet de comparer deux éléments entre deux sous la forme « a < b ».

1. Tri par sélection

Le principe de ce tri est d'aller « sélectionner » à chaque étape l'élément le plus petit du reste de la liste à trier et d'aller le mettre à la suite des éléments déjà triés.
Voici l'algorithme de tri par sélection, prenant pour argument une liste nommée simplement liste :

```
Pour i allant de 0 à n-2 Faire
    i_mini←i
    mini←liste[i]
    Pour j allant de i+1 à n-1 Faire
        Si liste[j] < mini Alors
            i_mini←j
            mini prend la valeur liste[j]
        FinSi
    Fin Pour
    échanger liste[i] et liste[i_mini]
Fin Pour
```

La boucle Pour la plus intérieure est celle qui recherche le minimum, ainsi que son indice, dans la partie non encore triée de la liste, de manière assez évidente (nous ne le démontrerons pas ici).
À mesure que l'on avance, les éléments de la liste de l'indice 0 à i sont triés, c'est ce que nous démontrerons.

Exercice 1
Coder cet algorithme dans le langage Python puis le tester (sur de petites listes).

Il est à noter que cet algorithme modifie la liste. Si on souhaite conserver la liste et obtenir en sortie une autre liste, il faut créer une copie de la liste, appliquer l'algorithme à cette copie et retourner la copie une fois triée.
Étudions maintenant les aspects théoriques de cet algorithme, à savoir la validité de l'algorithme puis son coût.

Pour la validité, on doit démontrer que l'algorithme termine et qu'il est correct.

- Tout d'abord, cet algorithme termine, puisque l'on a affaire à deux boucles Pour dont le nombre de passages est parfaitement déterminé et fini.
- Étudions alors la correction de l'algorithme. Pour cela, il nous faut un invariant de boucle (celui de la boucle principale).
  L'invariant est : « quelque soit i, la liste est une permutation de la liste initiale (c'est à dire qu'elle contient les mêmes éléments, dans un ordre éventuellement différent), la liste liste[0 :i] (indice i compris) est triée et tous les éléments de la liste liste[i+1 :n-1] sont supérieurs à tous les éléments de la liste liste[0 :i] ».
  Le fait que ce soit une permutation est évident, puisque le seul endroit où des éléments sont modifiés dans la liste, ils le sont en échangeant deux éléments de la liste.
  * Pour l'initialisation, on peut estimer qu'avant la boucle, la propriété n'a pas de sens puisque i n'existe pas. On démontre donc qu'après la première itération, la propriété est vraie. Après cette première itération, on a i=0, et la liste liste[0 :i] ne contient alors qu'un seul élément et est forcément triée. D'autre part, la valeur qu'elle contient est le minimum qui a été déterminé par la boucle Pour imbriquée, donc cette valeur est bien inférieure à tous les autres éléments de la liste.
  * Pour l'hérédité, on suppose que la propriété est vrai pour un i donné, on doit démontrer qu'elle le reste après une itération, pour i'=i+1. Or, on sait par hypothèse que les éléments de liste[0 :i] sont triés et que les éléments de liste[i+1 :n-1] sont tous supérieurs à ceux de liste[0 :i]. Comme on en prend le minimum, ce minimum est donc supérieur aux éléments de liste[0 :i], ce qui fait que la liste liste[0 :i'] est bien triée, et comme c'est le minimum de liste[i+1 :n-1], tous les éléments de liste[i'+1 :n-1] sont bien supérieurs à tous les éléments de liste[0 :i'].
    La propriété est donc bien héréditaire.

  On en conclut que la propriété est bien un invariant de boucle. Elle est donc vraie après la dernière itération, de quoi on déduit directement que la liste complète est alors triée, et comme c'est une permutation de la liste initiale, il s'agit bien de la liste triée voulue.

L'algorithme est donc valide.
Pour le coût, comme il y a deux boucles imbriquées, à l'intérieur desquels on fait des comparaisons (c'est le nombre de comparaisons que l'on compte pour déterminer le coût des algorithmes de tri), avec des bornes fixes pour la première (allant de 0 à $n-2$), et des bornes allant de i à $n-1$ pour l'autre, nous avons déjà vu que cela correspond à un coût quadratique en n, la longueur de la liste. Autrement dit, il y a un nombre de comparaisons de l'ordre de n^2, ce qui n'est pas très efficace. Il a l'avantage d'être facile à programmer, et à être suffisant pour de petites listes (disons jusqu'à 10^4 éléments).
Voir sur Wikipedia la page du tri par sélection qui en donne quelques animations.

2. Tri par insertion

Il s'agit ici de prendre les cartes à trier une par une, et de les « insérer » au bon en droit dans la partie triée de la liste. C'est une des méthodes naturelles du tri de cartes.
L'élément à inséré est appelé « clé », et il est inséré en décalant tous les éléments qui lui sont supérieurs dans la partie triée de la liste, parcourue de droite à gauche.
L'algorithme est le suivant :

```
Pour i allant de 0 à n-2 Faire
    k←i+1    (k pour key, indice de la clé)
    cle←liste[k]
    Tant que cle < liste[k-1] et k > 0 Faire
        liste[k]←liste[k-1]
        k prend la valeur k-1
    Fin Tant que
    liste[k]←cle
Fin Pour
```

Exercice 2
Coder cet algorithme dans le langage Python puis le tester (sur de petites listes).

Démontrons là aussi la validité de l'algorithme (sans entrer trop dans les détails).

- Pour la terminaison, on observe qu'une des boucles est une boucle Pour dont le nombre d'itérations est déterminé et fini. La boucle interne est une boucle Tant que. On doit donc trouver un variant. Le variant est tout simplement la variable k, qui décroît de 1 à chaque itération, et qui donc nécessairement deviendra négatif dans le cas où la première condition de la boucle est vérifiée, ce qui fait que la seconde condition fini par être fausse. Autrement dit il y a au plus i+1 itérations de la boucle Tant que.
- Étudions alors la correction de l'algorithme.
  L'invariant que nous utilisons pour la boucle Pour est : « pour chaque i, la liste est une permutation de la liste initiale et la liste liste[0 :i+1] est triée »
  Le fait que ce soit bien une permutation vient du processus d'insertion utilisé (la partie donnée par la boucle Tant que) : on « écrase » successivement, par la droite, les éléments de la liste par l'élément précédent, en commençant par la clé, mais on insère enfin la clé à sa place. Donc à la fin de la boucle Tant que, tous les éléments de la liste initiale se retrouvent bien dans la liste.
  * De même que pour l'algorithme précédent, on démontre l'initialisation en démontrant qu'après la première itération, la propriété est vraie.
    Au premier passage, la variable i vaut 0 et la clé est la valeur d'indice 1. Après la boucle Tant que, cette clé est insérée au bon endroit par rapport au premier élément. Autrement dit, la liste liste[0 :1] (indice 1 compris) est bien triée.
  * Si après l'itération pour une valeur i la propriété est vraie, alors l'élément liste[i+2] est dans l'itération suivante inséré (avec la boucle Tant que) au bon endroit dans la liste liste[0 :i+1] ou reste à sa place (c'est le cas si sa valeur est supérieure à celles de liste[0 :i+1]). Ainsi, dans tous les cas la liste liste[0 :i+2], autrement dit liste[0 :i'+1] (où i'=i+1), est bien triée.

  La propriété étant initialisée et héréditaire, on en déduit qu'elle est effectivement un invariant de boucle. Elle est donc vraie à la sortie de la boucle, d'où l'on déduit que la liste entière est triée.

Pour le coût de l'algorithme, il y a ici aussi deux boucles imbriquées, et nous avons indiqué le pire cas pour la seconde boucle. Ce pire cas est donc d'ordre n^2. Cependant, dans le meilleur cas, celui où la liste est déjà triée, il n'y a qu'une seule comparaison dans la seconde boucle, autrement dit une seule itération, et globalement l'algorithme devient alors linéaire.
En conséquence, pour une liste presque triée, cet algorithme est efficace. Le pire cas est celui où la liste est triée à l'envers, dans ce cas le coût est vraiment quadratique. En moyenne, le coût de cet algorithme est quadratique.
Voir sur Wikipedia la page du tri par insertion qui en donne quelques animations.

V. Algorithmes gloutons

Les termes « algorithme glouton » désignent une stratégie de résolution, consistant à construire une solution pas à pas (autrement dit de manière incrémentale, sans revenir en arrière), en faisant à chaque pas un choix local qui semble optimal.
Cette stratégie s'applique à des problèmes d'optimisation (où l'on cherche une solution optimale, selon un certain critère, à un problème donné) tels que ceux qui sont donnés plus bas. Il est à savoir que de manière générale, certains problèmes d'optimisation admettent des solutions gloutonnes, et d'autres pas (cela peut même dépendre des données du problème).
De plus, il n'est pas évident que l'accumulation des choix localement optimaux conduise nécessairement à une solution globalement optimale. En principe il est donc nécessaire de le démontrer, ce qui peut s'avérer très difficile, mais ce n'est pas l'objet en première (ni en terminale).
Dans certains cas, un algorithme glouton peut donner une approximation de la solution optimale.
Trouver un algorithme glouton peut être difficile, car il faut trouver le bon point de vue qui permet d'obtenir cette optimalité. Nous nous contenterons donc ici de les donner.

1. Rendu de monnaie

On se donne un système de pièces $i \in \{1, 5, 10, 20, 50\}$. L'objectif est de rendre une somme m de monnaie avec le moins de pièces possibles.
On considère bien sûr toutes les valeurs comme étant entières.
Une solution $S = \{i_1, i_2, \ldots, i_p\}$ est un multi-ensemble (il peut contenir des doublons) de pièces. La solution est optimale si p est minimal.
Une solution gloutonne consiste à choisir la plus grande pièce possible à chaque étape.

Exemple Pour $m = 89$, la solution gloutonne $50, 20, 10, 5, 1, 1, 1, 1$ est optimale.

Remarque La solution gloutonne n'est pas optimale pour tous les systèmes de monnaie.
Prenons par exemple $\{1, 4, 6\}$ et la somme $m = 8$.
La solution gloutonne $\{6, 1, 1\}$ est moins bonne que $\{4, 4\}$.
D'autre part, certains systèmes de pièces empêchent même qu'une solution existe (optimale ou non) quelque soit la somme m à rendre. Si le système contient 1, on est assuré d'avoir une solution (au pire, on ne rend que des pièces 1).

Exercice 3
Écrire une fonction Python pour le rendu de monnaie, prenant en paramètre une liste P pour le système de pièces et une somme m à rendre, et qui utilise la solution gloutonne. Cette fonction retourne une liste solution S, celle des pièces à utiliser pour rendre la monnaie.
Tester alors la fonction pour vérifier son bon fonctionnement sur plusieurs exemples (de systèmes de pièces et de somme à rendre).

2. Un problème de sac à dos

Il existe beaucoup de variantes de problèmes de sac à dos, allant de simple à compliqué dans leur donnée et dans leur résolution, qui n'admettent pas forcément de stratégie gloutonne optimale.
Nous en voyons ici un sous une forme simple, pour lequel la stratégie gloutonne n'est pas optimale.
On dispose de n objets $\{x_1, \ldots, x_n\}$ ayant chacun une valeur v_i et une masse m_i pour $1 \leqslant i \leqslant n$.

On possède un sac à dos ne pouvant supporter qu'une masse maximale M (évidemment, pour que le problème ait un intérêt, la somme totale des n objets, $m_1 + \cdots + m_n$ est supérieure à M).
On cherche le bon choix d'objets à mettre dans le sac à dos de sorte que la valeur totale des objets qu'il contient soit maximale.

Exercice 4

On note m la masse totale des objets choisis, et v la valeur totale de ces objets.
Sachant que chacun des n objets est soit choisi soit non choisi, on peut considérer qu'il existe des nombres c_i, valant respectivement chacun soit 1 soit 0, tels que :

$$m = c_1 m_1 + \cdots + c_n m_n \quad \text{et} \quad v = c_1 v_1 + \cdots + c_n v_n$$

1. Quelles sont les conditions à satisfaire pour m et v ?
2. On souhaite trier les objets par ordre de priorité décroissante.
   Qu'est-ce que cela signifie ?
3. Proposer un algorithme utilisant une stratégie gloutonne, à coder en Python, qui indique les objets à mettre dans le sac.
   La fonction Python prend deux arguments :
   - La liste des objets, sous forme d'une liste de couples (masse, valeur) et déjà triée selon l'ordre établi précédemment.
   - La masse maximale M.

   L'algorithme retourne la liste des c_i, indiquant donc si les objets sont pris ou non, ainsi que le poids mis dans le sac et la valeur totale.
4. Pour tester l'algorithme, on propose la liste suivante d'objets :

   [(14,126),(2,32),(5,20),(1,5),(6,18),(8,80)]

   avec un sac à dos de valeur $M = 15$
   On pourra tester différentes manières d'ordonner les objets, et voir les valeurs totales obtenues par la stratégie gloutonne.
   (On rappelle qu'il existe un moyen en Python de trier une liste selon une valeur calculée par une fonction : voir l'activité sur les données en tables)
   La solution optimale, que les diverses manières de trier les objets ne permettent pas d'obtenir par la méthode de résolution gloutonne, a une valeur totale de 132 $(32 + 20 + 80)$ pour une masse de $2 + 5 + 8 = 15 = M$.