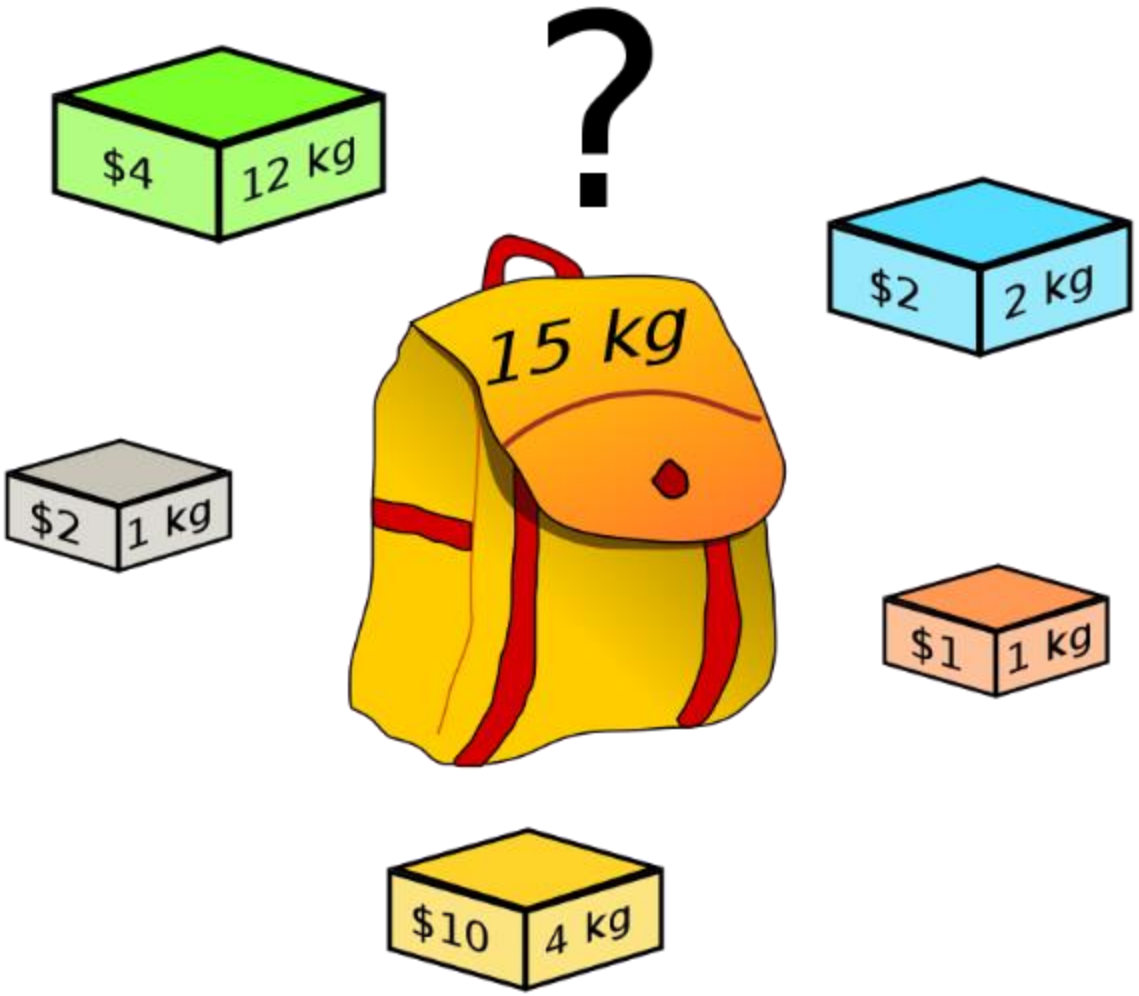
# Le problème du sac à dos



# Enoncé

Vous avez plusieurs objets possédant chacun un poids et une valeur.

Vous avez un sac qui peut supporter un certain poids maximum.

Quels objets faut-il mettre dans le sac de manière à maximiser la valeur totale du sac sans dépasser son poids maximal?

https://interstices.info/le-probleme-du-sac-a-dos/

# Historique

Ce problème fait partie des 21 problèmes **NP-complets** identifiés par Richard Karp en 1972.

Rappel: un problème NP  est un problème algorithmique tel qu'il est possible de vérifier « rapidement » si une solution candidate est bien solution. Par contre, on ne connaît pas d'algorithme permettant de trouver rapidement une solution générale.

On le retrouve:

- **dans la finance**: étant donné un certain montant d'investissement dans des projets, quels projets choisir pour que le tout rapporte le plus d'argent possible;

- **pour la découpe de matériaux**, afin de minimiser les pertes dues aux chutes;

- **dans le chargement de cargaisons** (avions, camions, bateaux…);

# Formulation mathématique

On caractérise chaque objet $O_i$ par son poids $p_i$ et sa valeur $v_i$.

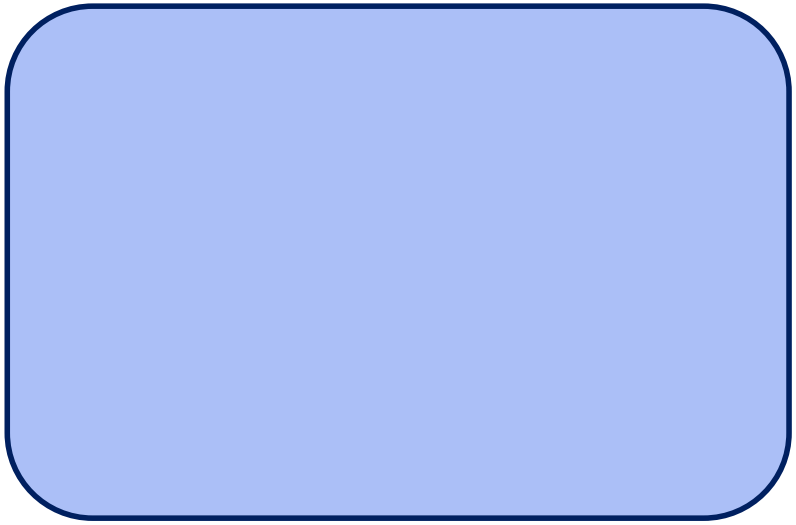
| Objet | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Poids (kg) | 12 | 4 | 2 | 1 | 1 |
| Valeurs (€) | 4 | 10 | 2 | 1 | 2 |

On définit une variable booléenne $q_i$ pour chaque objet tel que:

- $q_i = 0$ Si $O_i$ n'est pas dans le sac.
- $q_i = 1$ Si $O_i$ est placé dans le sac.

- Soit $P$ le poids maximal du sac et $V$ la valeur totale du sac.

Le problème du sac à dos revient à chercher les valeurs des $q_i$ telles que:



# Méthodes de résolution

- On va voir trois algorithmes pour tenter de résoudre ce problème:
  - La méthode gloutonne,
  - La méthode par force brute,
  - La méthode par programmation dynamique.

# Algorithme glouton (« greedy algorithm »)

Un **algorithme glouton** est un algorithme qui recherche la solution optimale d'un problème (**optimum global**) en effectuant à chaque étape le meilleur choix possible pour l'étape en cours (**optimum local**) sans jamais revenir sur ses choix.

# Algorithme glouton

- Ici, le meilleur choix possible à chaque étape est de choisir
- On calcule le ratio valeur/poids de chaque objet:

$$c_i = \frac{v_i}{p_i}$$

- On place les objets par valeurs de
- On remplit le sac ainsi jusqu'à la limite de poids du sac.

**Attention, rien ne dit que l'algorithme glouton donne la bonne solution à tous les coups!**

| Objet | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Poids (kg) | 12 | 4 | 2 | 1 | 1 |
| Valeurs (€) | 4 | 10 | 2 | 1 | 2 |
| Val/poids | 1/3 | 2,5 | 1 | 1 | 2 |

$P = 15\ kg$

- Algorithme glouton:

| Objet | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Poids (kg) | 13 | 12 | 8 | 10 |
| Valeurs (€) | 7 | 4 | 3 | 3 |
| Val/poids | 0,54 | 0,33 | 0,37 | 0,30 |

$P = 30\ kg$

- Algorithme glouton:

# Codage en Python

1) Pour stocker la liste des objets, on utilisera un dictionnaire. Le nom des objets sera les clés, les valeurs seront des listes de la forme : $[v_i, p_i]$

```
1 objets = {1:[4,12],2:[10,4],3:[2,2],4:[1,1],5:[2,1]}
```

2) On définit ensuite la fonction `ratio(objets)` qui renvoie une liste des ratios valeurs/poids pour chaque objet.

```
1 def ratio(objets):
2
3
4
5     return list_ratio
```

3) On modifie ensuite le dictionnaire pour ajouter dans chaque liste le ratio calculé à l'aide de la fonction `ajout_ratio(objets)`.

```
1  def ajout_ratio(objets):
2
3
4
5
6
7      return objets
```

Le dictionnaire a maintenant la forme:

```
{1: [4, 12, 0.3333333333333333], 2: [10, 4, 2.5], 3: [2, 2, 1.0],
```

4) Il faut maintenant trier la liste des objets afin qu'ils soient dans l'ordre décroissant des ratios. Pour cela, on va utiliser la fonction python `sorted`. L'aide de Python dit:

```
1 help(sorted)
```

```
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

Traduction:

On effectuera le tri sur la liste des tuples `(clé,valeur)` obtenu avec la commande `objets.items()`:

```
1 objets.items()
```

```
dict_items([(1, [4, 12, 0.3333333333333333]), (2, [10, 4, 2.5]), (3, [2, 2, 1.0]), (4, [1, 1, 1.
0]), (5, [2, 1, 2.0])])
```

On applique la fonction `sorted` en précisant le contenu de la fonction:

```
1 objets = sorted(objets.items(),key=ma_function,reverse=True)
```

```
1 def ma_fonction(liste):
2     return
```

5) Pour savoir si les objets sont pris dans le sac, on créé un dictionnaire appelé `objets_pris`:

```
1  objets_pris = {1:False,2:False,3:False,4:False,5:False}
```

Et on écrit ensuite une fonction `poids_sac` qui renvoie le poids du sac en fonction des objets qu'il contient:

```
1  def poids_sac(objets,objets_pris):
2
3
4
5
6      return p
```

6) Enfin, on écrit une fonction `sac_a_dos_glouton(objets_ord,Pmax)` qui prend en argument:

- la liste des objets ordonnée par ratio décroissant,
- `Pmax`, entier correspondant au poids maximal du sac

et qui renvoie:

- Le dictionnaire des objets effectivement pris dans le sac à dos,
- La valeur totale du sac

```
1  def poids_sac(objets_ord,Pmax):
2      p,v = 0,0 # poids et valeurs initiales
3      objets_pris=dict()
4      for x in objets_ord:
5          if                : # si, en rajoutant l'objet, on ne dépasse pas Pmax
6                              # noter l'objet comme pris
7                              # rajouter le poids de l'objet au sac
8                              # rajouter la valeur de l'objet au sac
9          else:
10                             # l'objet n'est pas pris
11     return (objets_pris,v)
```

# Algorithme par force brute

**Méthode par force brute**: on traite tous les cas possibles et on regarde à la fin quel est le meilleur des cas!

Avantage: ça marche à tous les coups!
Inconvénient: ça risque d'être très très long!

Pour 3 objets ayant un poids et une valeur, il faut tester combinaisons.
Pour 5 objets, il faut tester combinaisons.
Pour 10 objets, il faut tester combinaisons.

La complexité d'un tel algorithme est donc

# Codage en Python

Pour stocker la liste des objets, on utilise le même dictionnaire que précédemment:

```
1 objets = {1:[4,12],2:[10,4],3:[2,2],4:[1,1],5:[2,1]}
```

On appelle *paquetage* un choix de plusieurs objets dans le sac. Ce paquetage est codé en notation binaire: 01100 désigne, dans le cas où *n = 5,* un paquetage contenant les objets $O_2$ et $O_3$.

Pour chaque paquetage, il faut:

- Vérifier qu'il est valide (son poids doit être inférieur à *P*),
- Pouvoir calculer sa valeur.

1) On écrira tout d'abord une fonction `combinaison` prenant en argument un entier *n* et renvoyant la liste de toutes les chaînes composées de *n* 0 ou 1.

```
> combinaisons(3)
['000', '001', '010', '011', '100', '101', '110', '111']
> combinaisons(4)
['0000', '0001', '0010', '0011', '0100', '0101', '0110', '0111', '1000',
    '1001', '1010', '1011', '1100', '1101', '1110', '1111']
```

- Pour cela, on pourra utiliser la fonction `bin` de Python qui à un entier associe son écriture en base 2 sous la forme d'une chaîne de préfixe '`0b`':

```
> bin(0)
'0b0'
> bin(8)
'0b1000'
> bin(45)
'0b101101'
```

- Que renvoie `bin(76)` ?

```
def combinaisons(n):
    list_comb = []
    for i in range(...,...):
        ...
    return list_comb
```

2) On écrit ensuite une fonction `paq_to_sac` qui prend pour arguments un dictionnaire d'objets, une chaîne `paq` et un entier `pmax` correspondant au poids maximum du sac. La fonction renvoie `None` si le paquetage est trop lourd, sinon, elle renvoie un tuple contenant la liste des noms d'objets, le poids et la valeur des paquetages.

```
def paq_to_sac(objets,paq,pmax):
    sac,p,v = [],0,0 #initialisation
    for i in range(.........):
        if ...:
            ...
            ...
            ...
    if ...:
        return ...
    else:
        return ...
```

3) On désire obtenir maintenant par force brute la solution optimale. On écrit donc une fonction `sol_opt` qui prend pour arguments le dictionnaire des objets et un entier `pmax` correspondant au poids maximum du sac. La fonction affiche tous les tuples possibles renvoyés par la fonction `paq_to_sac` triés dans l'ordre décroissant des valeurs.

```
def ma_fonction_de_tri(liste)
    return ...

def sol_opt(objets,pmax):
    n = len(objets)
    l_comb = combinaisons(n)
    l_sol = [] #liste des solutions
    for paq in comb:
        s = paq_to_sac(.....................)
        if s:
            ..............................
    l_sol_tri = sorted(l_sol,key=ma_fonction_de_tri, reverse=True)
    ......................
    ......................
```