ITC2

Le problème du sac à dos

C. MULLAERT

Lycée Saint-Louis

Année 2025-2026

Le problème du sac à dos

On dispose de N objets, chacun ayant une masse m_i entière et une valeur v_i et un sac à dos pouvant contenir un poids maximal M. On cherche

$$P_{n,M} = \max \left\{ \sum_{i=1}^{n} x_i v_i,\ (x_1, ..., x_n) \in \{0,1\}^n \text{ tel que } \sum_{i=1}^{n} x_i m_i \leq M \right\}.$$

c'est-à-dire la valeur maximale que l'on peut mettre dans un sac à dos de capacité maximale M en utilisant les n premiers objets.

On peut également s'intéresser à une constitution optimale du sac à dos, c'est-à-dire à un n-uplet $(x_1, ..., x_n) \in \{0,1\}^n$ tel que

$$\sum_{i=1}^{n} x_i m_i \leq M \text{ et } \sum_{i=1}^{n} x_i v_i = P_{n,M}.$$

Une approche par force brute, c'est-à-dire en testant toutes les possibilités est à exclure car exponentielle en n.

Une approche gloutonne a été vue en première année, elle ne permet pas toujours d'obtenir la solution optimale.

Pour cela, à chaque étape, on met dans le sac à dos l'objet ayant le meilleur ratio valeur/poids ayant un poids inférieur à ce qu'il est encore possible de mettre dans le sac à dos.

```
def SacADos(Lval, Lpoids,Max):
    n=len(Lval)
    if min(Lpoids)>Max :
        return(n*[0])
    #Recherche de l'objet le plus rentable
    #et de poids<=Max
    qmax=0
    for k in range(n) :
        pk=Lpoids[k]
        vk=Lval[k]
        if pk<=Max and (vk/pk)>qmax:
            i=k
            qmax=vk/pk
    pi=Lpoids[i]
    Lpoids[i]=Max+1
    #L'objet est mis à Max+1 pour ne plus être pris
    L= SacADos(Lval, Lpoids,Max-pi)
    L[i]=1
    return(L)
```

Pour gagner en complexité, on peut également trier les objets en fonction de leur ratio valeur/poids en adaptant l'un des algorithmes de tri : tri bulle, tri par insertion, tri par sélection, tri fusion, tri rapide, tri par comptage.

Le tri bulle consiste à parcourir la liste et à comparer les éléments consécutifs et, lorsqu'ils ne sont pas dans le bon ordre, on les échange. Après ce premier passage, le plus grand élément de la liste est à la fin. On recommence le nombre de fois nécessaire.

```
def TriBulle(L) :
    n=len(L)
    p=0 #numéro de passage
    c=1 #nombre d'échanges de ce passage
    while c>0 :
        c=0
        for k in range(n-p-1):
            if L[k]>L[k+1] :
                L[k],L[k+1] =L[k+1],L[k]
                c+=1
        p+=1
    return(L)
```

Il s'agit d'un tri par comparaison, en place, stable (grâce à l'inégalité stricte).

La complexité est quadratique dans le pire des cas et linéaire dans le meilleur des cas grâce au compteur.

Néanmoins, il est très peu utilisé en pratique.

Le tri par insertion consiste à déplacer le deuxième élément de sorte que les deux premiers éléments soient bien ordonnés puis à recommencer. À chaque itération i suivante, on part d'une liste dont les i premiers éléments sont bien triés, et l'on déplace le $(i+1)$-ème élément pour l'insérer à sa place parmi tous ceux qui le précèdent.

```python
def Triinsert(L) :
    n=len(L)
    for i in range(1,n):
        val=L[i]
        # On cherche la place de L[i]
        # et on décale les valeurs précédentes trop grandes
        j=i-1
        while j>=0 and L[j]>val :
            L[j+1]=L[j]
            j=j-1
        L[j+1]=val
```

Il s'agit d'un tri par comparaison, en place, stable, quadratique dans le pire des cas, linéaire dans le meilleur des cas.

Cette fonction ne renvoie rien elle agit par effet de bord.

Voici une version récursive :

```
def TriinsertRec(L):
    n=len(L)
    if n<= 1:
        return(L)
    val=L[-1]
    M=TriinsertRec(L[:-1])+[val]
    j = n-2
    while j >= 0 and M[j] > val:
        M[j+1] = M[j]
        j = j-1
    M[j+1] = val
    return(M)
```

Le tri est toujours stable mais il a l'inconvénient de ne pas être en place car on recopie des listes. Il est de complexité quadratique.

Voici une version récursive en place :

```
def TriinsertRec2(L) :
    def insertionRec(k,x) :
        # insère l'élément x dans la liste L[0:k]
        # supposée triée
        if k>0 and x< L[k-1] :
            L[k]=L[k-1]
            insertionRec(k-1,x)
        else :
            L[k]=x

    def TriJusque(j) :
        if j>1 :
            TriJusque(j-1)
            insertionRec(j-1,L[j-1])

    TriJusque(len(L))
```

On peut améliorer la complexité en recherchant l'indice où insérer le nouvel élément a de façon dichotomique. Pour que le tri soit stable on cherche, lorsqu'on insère a dans $L[0:k]$, un indice $d \in [\![1, k-1]\!]$, s'il existe, tel que $L[d-1] <= a < L[d]$.

```
def Dicho(k,a,L) :
        if a<L[0] :
            return 0
        if a>=L[k-1] :
            return k
        g=0
        d=k
        while d-g>1 : #invariants : g<=d et L[g]<=a<L[d]
            m=(g+d)//2
            if L[m]<=a :
                g=m
            else :
                d=m
        return d
```

```
def TriInsertDicho(L):
    for k in range(1,len(L)):
        a=L[k]
        ind=Dicho(k,a,L)
        for j in range(k,ind,-1) :
            L[j],L[j-1]=L[j-1],L[j]
```

Le nombre de comparaisons est alors semi-linéaire c'est-à-dire en $O(n \ln n)$, le nombre d'échanges reste quadratique dans le pire des cas.

Le tri par sélection consiste à placer le plus petit élément de la liste en première position, le plus petit des autres éléments en seconde position, et ainsi de suite.

```
def Triselection(L):
    n=len(L)
    for k in range(n-1) :
# On trouve le plus petit parmi les n-k derniers éléments
        m=L[k]
        imin=k
        for j in range(k+1,n):
            if L[j]<m :
                m=L[j]
                imin=j
        L[k],L[imin]=L[imin],L[k]
```

On retiendra qu'il s'agit d'un tri avec un nombre de comparaisons quadratique mais un nombre d'échanges linéaire, ce qui pourrait être utile si l'on compare autres chose que des entiers et si la comparaison est moins coûteuse que l'échange.

La version itérative précédente est en place et stable.

Une version récursive naïve nécessite le recopiage de listes

```
def TriselectionRec(L):
    n=len(L)
    if n<=1 :
        return(L)
    imin=0
    vmin=L[0]
    for k in range (1,n) :
        if L[k]<vmin :
            imin=k
            vmin=L[k]
    L2=TriselectionRec([L[k] for k in range(n) if k!=imin])
    return ([L[imin]]+L2)
```

mais on peut écrire une version récursive en place :

```
def TriselectionRec2(L):
    n=len(L)
    def aux(j) :
#place en position j le  plus petit des éléments de L[j:]
        imin=j
        vmin=L[j]
        for k in range (j+1,n) :
            if L[k]<vmin :
                imin=k
                vmin=L[k]
        L[j],L[imin]=L[imin],L[j]
    def TriJusque(j) :
        if j==0 :
            aux(0)
        else :
            TriJusque(j-1)
            aux(j)
    TriJusque(n-1)
```

Le tri fusion repose sur les deux idées suivantes

- si on dispose de deux listes déjà triées, il est facile de les fusionner en une seule liste triée, en sélectionnant successivement le premier élément de l'une ou de l'autre, selon celui qui est le plus petit ;
- une liste de longueur 0 ou 1 est déjà triée.

Il s'agit d'un algorithme naturellement récursif qui fonctionne en coupant la liste à trier en deux parties à peu près égales, qu'on coupe à leur tour en deux et ainsi de suite. Au bout d'un certain nombre d'appels récursifs, toutes les listes résultantes sont de longueur 0 ou 1, donc triées : il ne reste qu'à les fusionner en remontant la pile des appels.

```
def Fusion(L1,L2):
    L=[]
    n1=len(L1)
    n2=len(L2)
    i=0
    j=0
    while i<n1 and j<n2 :
        if L1[i]<= L2[j] :
            L+=[L1[i]]
            i=i+1
        else :
            L+=[L2[j]]
            j=j+1
    if i<len(L1) :
        return(L+L1[i:])
    return(L+L2[j:])
```

```
def TriFusionRec(L) :
    n=len(L)
    if n<=1 :
        return(L)
    L1=TriFusionRec(L[:n//2])
    L2=TriFusionRec(L[n//2:])
    return FusionRec(L1,L2)
```

On retiendra qu'il s'agit d'un tri qui n'est pas en place et dont la complexité est en $n \log n$ où n représente la longueur de la liste à trier.

Le nombre de comparaisons ne peut pas être amélioré pour un algorithme par comparaisons

Le tri par pivot repose sur l'idée suivante : on fixe un des éléments a de la liste à trier (par exemple son premier élément pour avoir la stabilité), qui est le "pivot".

On partitionne ensuite les autres éléments de la liste en deux listes, celle des éléments strictement inférieurs à a et les autres. On réitère ce procédé sur ces deux liste.

Il s'agit donc, comme le tri par fusion, d'un algorithme naturellement récursif.

```
def Partitionne(L):
    Lg=[]
    Ld=[]
    p=L[0]
    for k in range(1,len(L)):
        if L[k]<p :
            Lg+=[L[k]]
        else :
            Ld+=[L[k]]
    return(Lg,Ld)

def TriRapide(L) :
    if len(L)<=1:
        return(L)
    Lg,Ld=Partitionne(L)
    return(TriRapide(Lg)+[L[0]]+TriRapide(Ld))
```

On retiendra que le tri rapide est statistiquement, de loin, le tri le plus rapide en moyenne (d'où son surnom), il est en $n \log n$ où n représente la longueur de la liste à trier, ce qui est optimal pour un tri par comparaisons.

Néanmoins pour une liste triée, il est de complexité quadratique.

Enfin le tri par comptage ou dénombrement est utile lorsque les valeurs de la liste sont par exemple des entiers entre 0 et M.

```
def TriComptage(L, M):
    occ = [0] * (M+1)
    for i in L:
        occ[i] += 1
    s = 0
    for i in range(M+1):
        for j in range(occ[i]):
            L[s] = i
            s += 1
```

Si $M = O(len(L))$, on a un algorithme de complexité linéaire.

Ce problème possède une sous-structure optimale car l'on a la relation de récurrence :

$$P_{k,m} = \begin{cases} P_{k-1,m} & \text{si } m_k > m \\ \max\left(P_{k-1,m}, v_k + P_{k-1,m-m_k}\right) & \text{sinon} \end{cases}$$

On va donc procéder en utilisant la programmation dynamique.

La **mémoïsation** consiste à modifier la version récursive naïve suivante :

```
def sacadosR(Lm,Lv,M):
#Lm est la liste des masses, Lv celle des valeurs
#et M la masse maximale
    def aux(k,m) :
    #renvoie la valeur maximale pour un sac de contenance
    #maximale m et k objets
        if k==0:
            return 0
        mk=Lm[k-1]
        vk=Lv[k-1]
        if mk>m :
            return aux(k-1,m)
        return max(aux(k-1,m), aux(k-1,m-mk)+vk)
    return aux(len(Lm),M)
```

en stockant les valeurs déjà calculées :

```
def sacadosM(Lm,Lv,M):
    d={}
    def aux(k,m) :
        if (k,m) in d :
            return d[(k,m)]
        if k==0 :
            d[(k,m)]=0
            return 0
        mk=Lm[k-1]
        vk=Lv[k-1]
        if mk>m :
            p=aux(k-1,m)
        else :
            p= max(aux(k-1,m), aux(k-1,m-mk)+vk)
        d[(k,m)]=p
        return p
    return aux(len(Lm),M)
```

On peut également remplir une liste de listes **de bas en haut** :

```
def sacados_Asc(Lm,Lv,M):
    n=len(Lm)
    T=[(M+1)*[0] for i in range(n+1)]
    for k in range(1,n+1):
        for m in range(0,M+1):
            mk=Lm[k-1]
            vk=Lv[k-1]
            if mk>m :
                T[k][m]=T[k-1][m]
            else :
                T[k][m]=max(T[k-1][m],T[k-1][m-mk]+vk)
    return T[n][M]
```

Déterminons une composition optimale du sac à dos

```
def sacados_dico(Lm,Lv,M):
    d={}
    def aux(k,m) :
        if (k,m) in d :
            return d[(k,m)]
        if k==0:
            d[(k,m)]=0
            return 0
        mk=Lm[k-1]
        vk=Lv[k-1]
        if mk>m :
            p=aux(k-1,m)
        else :
            p= max(aux(k-1,m), aux(k-1,m-mk)+vk)
        d[(k,m)]=p
        return p
    aux(len(Lm),M) #on remplit le dictionnaire
    return d
```

```
def sacados_composition_dico(Lm,Lv,M):
    d=sacados_dico(Lm,Lv,M)
    Sac=[]
    M2=M
    for k in range(len(Lm),0,-1):
        if d[(k,M2)]>d[(k-1,M2)]:
            Sac=[1]+Sac
            M2-=Lm[k-1]
        else :
            Sac=[0]+Sac
    return Sac
```

L'ajout d'un élément en début de liste n'est pas adapté à cette structure. On peut utiliser le type deque.

```
def sacados_composition_dico(Lm,Lv,M):
    d=sacados_dico(Lm,Lv,M)
    Sac=deque()
    M2=M
    for k in range(len(Lm),0,-1):
        if d[(k,M2)]>d[(k-1,M2)]:
            Sac.appendleft(1)
            M2-=Lm[k-1]
        else :
            Sac.appendleft(0)
    return list(Sac)
```

ou plus simplement :

```
def sacados_composition_dico(Lm,Lv,M):
    d=sacados_dico(Lm,Lv,M)
    n=len(Lm)
    Sac=[0]*n
    M2=M
    for k in range(n,0,-1):
        if d[(k,M2)]>d[(k-1,M2)]:
            Sac[k-1]=1
            M2-=Lm[k-1]
        else :
            Sac[k-1]=0
    return Sac
```

On peut également éviter de refaire des comparaisons en stockant l'alternative choisie

```
def sacados_dico2(Lm,Lv,M):
    d,r={},{}
    def aux(k,m) :
        if (k,m) in d :
            return d[(k,m)]
        if k==0:
            d[(k,m)]=0
            return 0
        mk,v_k = Lm[k-1],Lv[k-1]
        if mk>m :
            p=aux(k-1,m)
        else :
            p1,p2 = aux(k-1,m),aux(k-1,m-mk)+vk
            if p1<=p2:
                r[(k,m)],p = 1,p2
            else :
                p=p1
        d[(k,m)]=p
        return p
    aux(len(Lm),M) #on remplit le dictionnaire
    return r
```

```
def sacados_composition_dico2(Lm,Lv,M):
    d=sacados_dico2(Lm,Lv,M)
    n=len(Lm)
    Sac=[0]*n
    M2=M
    for k in range(n,0,-1):
        if (k,M2) in r:
            Sac[k-1]=1
            M2-=Lm[k-1]
        else :
            Sac[k-1]=0
    return Sac
```

On peut de même adapter la version ascendante.

Il est surtout intéressant de stocker l'alternative lorsque la retrouver nécessite un nombre de comparaison qui augmente avec la taille du problème.

```
def sacados_composition_Asc(Lm,Lv,M):
    n=len(Lm)
    T=[(M+1)*[0] for i in range(n+1)]
    for k in range(1,n+1):
        for m in range(0,M+1):
            mk=Lm[k-1]
            vk=Lv[k-1]
            if mk>m :
                T[k][m]=T[k-1][m]
            else :
                T[k][m]=max(T[k-1][m],T[k-1][m-mk]+vk)
    Sac,M2=[0]*n,M
    for k in range(n,0,-1):
        if T[k][M2]>T[k-1][M2]:
            Sac[k-1]=1
            M2-=Lm[k-1]
    return Sac
```