SacaDos_projet_bloc2_vfinal

June 27, 2019

1 Problème du sac à dos

1.1 0. Prérequis

Cette activité aura lieu après avoir abordé l'ensemble des algorithmes de tris du programme.

1.2 1. Introduction

Voir diaporama.

L'objectif est que les élèves manipulent, expliquent leurs démarches et découvrent eux-mêmes les algorithmes à programmer.

1.2.1 Déclaration des variables et création d'une liste d'objets aléatoires

Le code suivant propose une création aléatoire d'une liste de n objets permettant de tester les codes qui suivent.

```
In [1]: import random as rd

        w = 30 # Poids maximum contenant dans le sac
        n = 20 # nombre d'objets disponibles
        vmax = 10 # valeur maximale d'un objet
        pmax = 10 # poids maximal d'un objet

        objets = [(rd.randrange(vmax)+1, rd.randrange(pmax)+1) for i in range(n)]
        print(objets)
```

```
[(6, 9), (9, 4), (7, 3), (4, 1), (7, 1), (10, 1), (8, 6), (6, 9), (10, 6), (3, 7), (2, 10), (2,
```

1.3 2. Algorithme de force brute

On teste toutes les combinaisons possibles. Nous avons choisi d'identifier une combinaison de n objets par un nombre en binaire à n chiffres, chaque chiffre indiquant si l'objet correspondant est sélectionné dans la combinaison ou pas.

```
In [2]: def forceBrute(objets,w):
            """
```

```
    Résolution du problème du sac à dos par force brute
    objets est une liste d'objets du type (poids,valeur)
    w est le poids maximum entrant dans le sac
    Un nombre en binaire permet d'indiquer quels objets sont sélectionnés.
    """
    n = len(objets)
    valeurmax = 0
    combinaisonmax = 0
    for combinaison in range(2**n):
        poids = 0
        valeur = 0
        combiBinaire = bin(combinaison)[2:]
        combiBinaire = (n-len(combiBinaire))*'0'+combiBinaire # avec n chiffres
        for j in range(n):
            if combiBinaire[j]=='1':
                poids += objets[j][1]
                if poids > w:
                    break
                valeur += objets[j][0]
        else:
            if valeur > valeurmax:
                valeurmax = valeur
                combinaisonmax = combiBinaire
    contenuSac = []
    for j in range(n):
        if combinaisonmax[j]=='1':
            contenuSac.append(objets[j])
    return valeurmax, contenuSac
```

```
In [3]: print(forceBrute(objets,w))
```

```
(80, [(9, 4), (7, 3), (4, 1), (7, 1), (10, 1), (8, 6), (10, 6), (7, 1), (2, 1), (10, 2), (6, 4)]
```

Q1. Quelle est, en fonction de n, la complexité de l'algorithme précédent ?

Réponse

2^n combinaisons ; pour chaque combinaison, les opérations effectuées (conversion en binaire, ajout d'au plus n zéros, une boucle contenant des opérations en temps constant) sont au plus en temps linéaire.

La complexité de cet algorithme est donc en $O(2^n \times n)$.

Q2. Estimer le temps de calcul de l'algorithme de force brute si on dispose de 50 objets ? 100 objets ?

Réponse

Le nombre d'opérations élémentaires est de l'ordre de $2^{50} \times 50 \approx 5 \times 10^{16}$.

Un ordinateur actuel réalise de l'ordre de 10^9 opérations élémentaires par seconde (fréquence de l'ordre du Ghz).

Il faudra donc de l'ordre de 5×10^7 secondes, soit environ 2 ans.

Remarque : il s'agit juste d'un ordre de grandeur puisque ces calculs sont effectués à une constante multiplicative près.

Avec 100 objets, il faut de l'ordre de 10^{32} opérations, soit 10^{23} secondes, environ un million de millards d'années !

1.4 3. Exemples d'algorithmes gloutons

1.4.1 Exemple 1

On remplit le sac à dos en mettant d'abord les objets de plus grande valeur, jusqu'à ce que plus aucun objet ne rentre dans le sac.

Q3. Programmer cet algorithme. On pourra utiliser la fonction tri donnée, ou la faire reprogrammer par les élèves ou utiliser la fonction Python *sorted*.

```
In [4]: """
        Tri rapide avec pivot aléatoire
        A remplacer par un tri sélection/insertion en première,
        ou utiliser la fonction sorted() de Python
        """
        def tri(lst):
            if len(lst) <= 1:
                return lst
            indice_pivot = rd.randrange(len(lst))
            pivot = lst[indice_pivot]
            lst1 = []
            lst2 = []
            for i in range(indice_pivot):
                x = lst[i]
                if x<pivot: lst1.append(x)
                else: lst2.append(x)
            for i in range(indice_pivot+1, len(lst)):
                x = lst[i]
                if x<pivot: lst1.append(x)
                else: lst2.append(x)
            return tri(lst1)+[pivot]+tri(lst2)
```

```
In [5]: def glouton1(objets,w):
            """
            Résolution du problème du sac à dos
            On met en priorité les objets de plus grosse valeur
            """
            contenuSac = []
            valeur = 0

            return valeur, contenuSac
```

Réponse

```
In [6]: def glouton1(objets,w):
            """
            Résolution du problème du sac à dos
```

```
            On met en priorité les objets de plus grosse valeur
            """
            objetsTries = tri(objets)
            contenuSac = []
            valeur = 0
            poids = 0
            while poids < w and objetsTries:
                v, p = objetsTries.pop()
                if poids + p <= w:
                    valeur += v
                    poids += p
                    contenuSac.append((v,p))
            return valeur, contenuSac
```

```
In [7]: glouton1(objets,w)
```

```
Out[7]: (62, [(10, 6), (10, 2), (10, 1), (9, 4), (8, 10), (8, 6), (7, 1)])
```

Q5. Quelle est la complexité de cet algorithme ?

Réponse La complexité de l'algorithme est égale à celle tri utilisé, $O(n \times ln(n))$ pour le tri rapide et $O(n^2)$ pour les tris utilisés par les élèves en première (sélection et insertion), en supposant qu'on n'est pas dans le meilleur des cas ... le reste de l'algorithme est en temps linéaire (un parcours de liste) donc négligeable devant le tri.

Q6. Comparer les deux algorithmes (force brute et glouton1).

Réponse L'algorithme glouton est beaucoup plus rapide mais ne renvoie pas toujours la solution optimale. On peut évaluer son pourcentage de succès, qui doit évoluer selon les paramètres, notamment le nombre d'objets n.

1.4.2 Exemple 2

On remplit le sac à dos en mettant d'abord les objets avec le meilleur rapport valeur/poids, jusqu'à ce que plus aucun objet ne rentre dans le sac.

Q8. Programmer cet algorithme.

```
In [8]: def glouton2(objets,w):
            """
            Résolution du problème du sac à dos
            On met en priorité les objets de meilleur rapport valeur/poids
            """
            contenuSac = []
            valeur = 0

            return valeur, contenuSac
```

Réponse

```
In [9]: def glouton2(objets,w):
            """
            Résolution du problème du sac à dos
```

```
        On met en priorité les objets de meilleur rapport valeur/poids
        """
        objets2 = [(v/p, v, p) for (v, p) in objets]
        objetsTries = tri(objets2)
        contenuSac = []
        valeur = 0
        poids = 0
        while poids < w and objetsTries:
            t, v, p = objetsTries.pop()
            if poids + p <= w:
                valeur += v
                poids += p
                contenuSac.append((v,p))
        return valeur, contenuSac
```

```
In [10]: glouton2(objets,w)
```

```
Out[10]: (80,
          [(10, 1),
           (7, 1),
           (7, 1),
           (10, 2),
           (4, 1),
           (7, 3),
           (9, 4),
           (2, 1),
           (10, 6),
           (6, 4),
           (8, 6)])
```

Q9. Comparer les trois algorithmes.

Réponse L'algorithme *glouton*2 est de même complexité que *glouton*1 puisqu'on rajoute juste une construction de liste en temps linéaire avant le tri.

*glouton*2 est généralement plus efficace que *glouton*1 (voir plus loin).

Q10. Ecrire une fonction *glouton*3 renvoyant le meilleur résultat des deux fonctions *glouton*1 et *glouton*2.

Réponse

```
In [11]: def glouton3(objets,w):
             res1 = glouton1(objets,w)
             res2 = glouton2(objets,w)
             return res1 if res1[0] > res2[0] else res2

         glouton3(objets,w)
```

```
Out[11]: (80,
          [(10, 1),
           (7, 1),
           (7, 1),
```

```
(10, 2),
(4, 1),
(7, 3),
(9, 4),
(2, 1),
(10, 6),
(6, 4),
(8, 6)])
```

Q11. Quelle est la complexité de l'algorithme *glouton*3 **?**
Réponse Même complexité que *glouton*1 et *glouton*2.

1.5 4. Programmation dynamique (cours de terminale)

1.6 4.1. Présentation

Le problème du sac à dos possède la propriété de sous-structure optimale, c'est à dire que l'on peut construire la solution optimale du problème à i variables à partir du problème à $i-1$ variables.

1.6.1 Propriété :

Soit $F_i(c)$ le meilleur coût pour remplir un sac de poids c avec les i premiers objets. On peut montrer par récurrence :

$$F_i(c) = \max\{F_{i-1}(c), F_{i-1}(c-p_i) + v_i\}$$

En effet, l'ajout d'un nouvel objet augmente ou non le meilleur coût :

- soit l'objet n'est pas rajouté dans le sac. On a alors, $F_i(c) = F_{i-1}(c)$
- soit $p_i \leqslant c$. On a alors, $F_i(c) = \max(F_{i-1}(c), F_{i-1}(c-p_i) + v_i)$

1.7 Exemple

i	$amp;1$	$amp;2$	$amp;3$	$amp;4$	$amp;5$
v_i	$amp;15$	$amp;18$	$amp;20$	$amp;12$	$amp;6$
p_i	$amp;3$	$amp;4$	$amp;5$	$amp;4$	$amp;2$

On fixe une limite de poids à 8 kg

	$amp;p$	$amp;0$	$amp;1$	$amp;2$	$amp;3$	$amp;4$	$amp;5$	$amp;6$	$amp;7$	$amp;8$
0	$amp;0$	$amp;0$	$amp;0$	$amp;0$	$amp;0$	$amp;0$	$amp;0$	$amp;0$	$amp;0$	$amp;0$
1	$amp;3$	$amp;0$	$amp;0$	$amp;0$	$amp;15$	$amp;15$	$amp;15$	$amp;15$	$amp;15$	$amp;15$
2	$amp;4$	$amp;0$	$amp;0$	$amp;0$	$amp;15$	$amp;18$	$amp;18$	$amp;18$	$amp;33$	$amp;33$
3	$amp;5$	$amp;0$	$amp;0$	$amp;0$	$amp;15$	$amp;18$	$amp;20$	$amp;20$	$amp;33$	$amp;35$
4	$amp;4$	$amp;0$	$amp;0$	$amp;0$	$amp;15$	$amp;18$	$amp;20$	$amp;20$	$amp;33$	$amp;35$
5	$amp;2$	$amp;0$	$amp;0$	$amp;6$	$amp;15$	$amp;18$	$amp;21$	$amp;24$	$amp;33$	$amp;35$

1.7.1 4.2. Activité manuelle

Q12. Exercice

Tester sur papier l'algorithme précédent avec un sac à dos de contenance maximale égale à 10 kg dans les deux cas suivants :

1. objets = [(3, 2), (8, 10), (2, 2), (8, 1), (4, 6), (6, 6)]
2. objets = [(5, 3), (9, 2), (10, 5), (6, 4), (7, 1), (9, 3)]

Les objets sont au format (valeur, poids).

1.7.2 4.3 Programmation

Q13. Programmer en Python, en l'adaptant aux variables utilisées dans les fonctions précédentes, l'algorithme de programmation dynamique donné sur la page Wikipedia.

Réponse

```
In [12]: def dynamique(objets,w):
             """
             Résolution du problème du sac à dos
             par programmation dynamique
             """
             n = len(objets)
             tab = [ [0]*(w + 1) for i in range(n + 1)]
             for i in range(n):
                 for c in range(w + 1):
                     v, p = objets[i]
                     if c >= p:
                         tab[i + 1][c] = max(tab[i][c], tab[i][c-p] + v)
                     else:
                         tab[i + 1][c] = tab[i][c]
             return tab
         objets=[(15,3),(18,4),(20,5),(12,4),(6,2)]
         dynamique(objets,8)

Out[12]: [[0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 15, 15, 15, 15, 15, 15],
          [0, 0, 0, 15, 18, 18, 18, 33, 33],
          [0, 0, 0, 15, 18, 20, 20, 33, 35],
          [0, 0, 0, 15, 18, 20, 20, 33, 35],
          [0, 0, 6, 15, 18, 21, 24, 33, 35]]
```

Q14. En utilisant les tableaux réalisés dans la question Q12, déterminer dans chacun des deux cas le contenu du sac à dos réalisant la valeur maximale. Expliquer votre démarche.

Q15. Écrire une fonction permettant de reconstituer le contenu du sac à dos correspondant à la solution optimale à partir du tableau renvoyé par la fonction *dynamique*.

Réponse

```
In [13]: def solutionDynamique(tab,objets):
             """
             Reconstitue la liste d'objets à partir du tableau
             renvoyé par la fonction dynamique
             """
             i = len(tab) - 1  # égal à len(objets)
             j = len(tab[0]) - 1
```

```
        contenuSac = []
        valeur = tab[i][j]
        while i > 0:
            if tab[i - 1][j] == tab[i][j]:
                pass # l'objet i n'est pas dans le sac
            else:
                v, p = objets[i - 1]
                contenuSac.append((v, p))
                j = j - p
            i = i - 1
        return valeur,contenuSac
```

```
In [14]: solutionDynamique( dynamique(objets,w), objets)
```

```
Out[14]: (71, [(6, 2), (12, 4), (20, 5), (18, 4), (15, 3)])
```

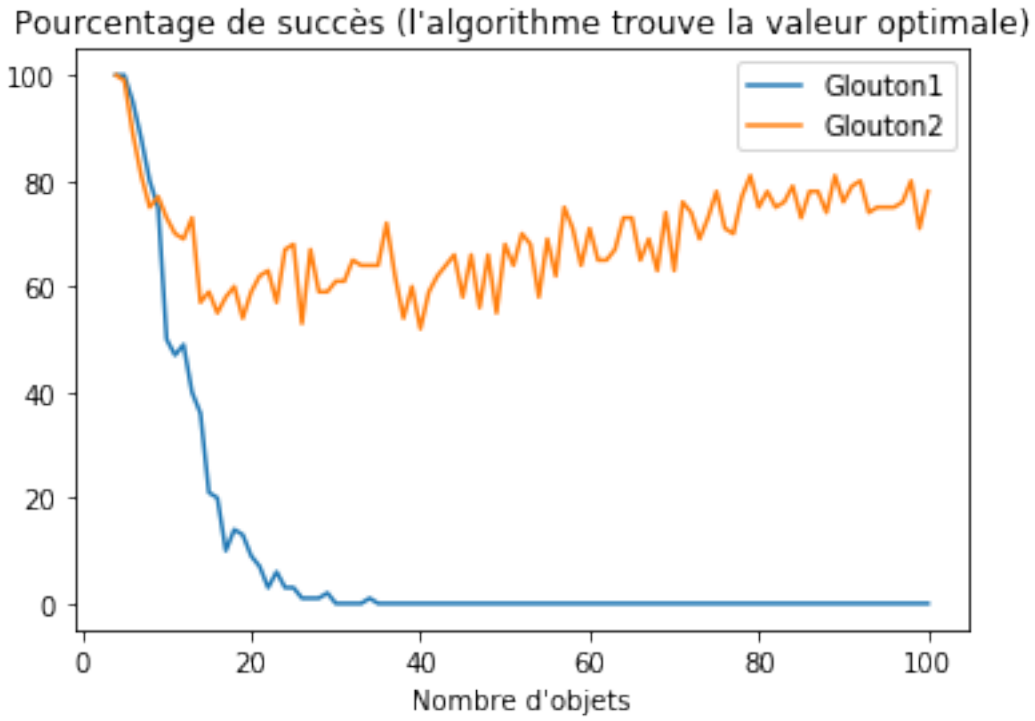
Q16. Quelle est la complexité de l'algorithme de programmation dynamique ?

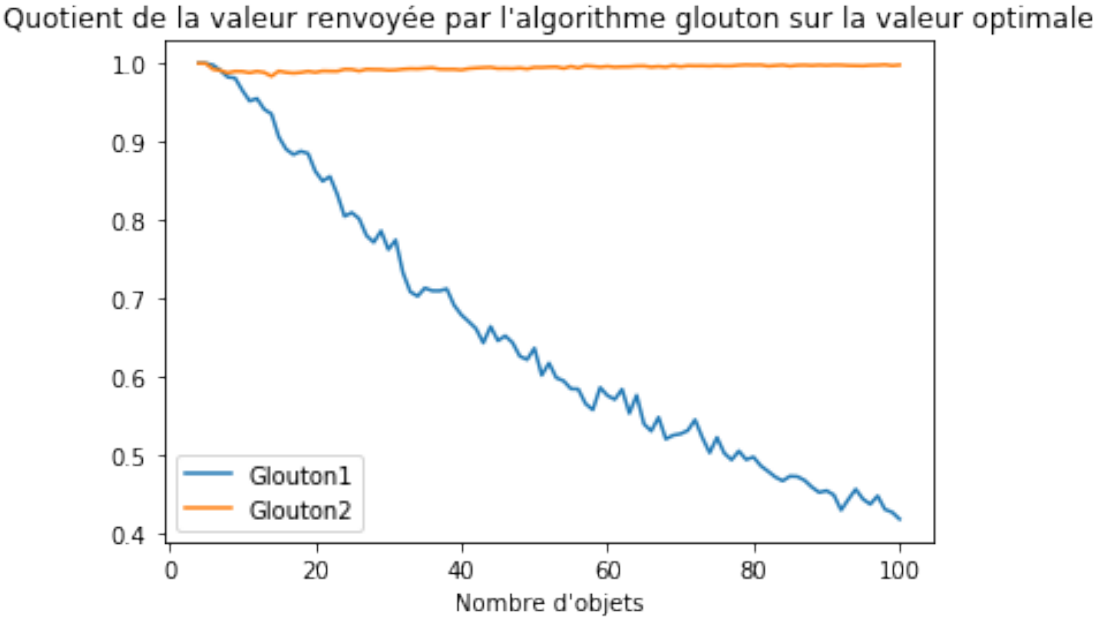
Réponse La fonction *dynamique* a une complexité égale au nombre de cases du tableau, donc en $O(w \times n)$. La fonction *solutionDynamique* a une complexité en $O(n)$ (une boucle). La complexité totale de cet algorithme est donc en $O(w \times n)$.

1.8 5. Tests d'efficacité des algorithmes gloutons

```
In [16]: import matplotlib.pyplot as plt

        def comparaison(n, w, vmax=15, pmax=10):
            objets = [(rd.randrange(vmax)+1, rd.randrange(pmax)+1) for i in range(n)]
            valeurOptimale = dynamique(objets,w)[-1][-1]
            valeurGlouton1 = glouton1(objets,w)[0]
            valeurGlouton2 = glouton2(objets,w)[0]
            return valeurGlouton1/valeurOptimale, valeurGlouton2/valeurOptimale

        def plusieursComparaisons(nb, n, w, vmax=15, pmax=10):
            lstglouton1 = []
            lstglouton2 = []
            for i in range(nb):
                t1, t2 = comparaison(n,w,vmax,pmax)
                lstglouton1.append(t1)
                lstglouton2.append(t2)
            moyGlouton1 = sum(lstglouton1)/len(lstglouton1)
            moyGlouton2 = sum(lstglouton2)/len(lstglouton2)
            nbOptGlouton1 = sum(x==1.0 for x in lstglouton1)/nb*100
            nbOptGlouton2 = sum(x==1.0 for x in lstglouton2)/nb*100
            """
            print("Sur",nb,"simulations, l'algorithme glouton1 renvoie une valeur moyenne égale
            print("Sur",nb,"simulations, l'algorithme glouton2 renvoie une valeur moyenne égale
            print("Sur",nb,"simulations, l'algorithme glouton1 trouve la valeur optimale",nbOpt
            print("Sur",nb,"simulations, l'algorithme glouton2 trouve la valeur optimale",nbOpt
            """
```

```
    return moyGlouton1, moyGlouton2, nbOptGlouton1, nbOptGlouton2

def courbeGloutons(nb, nmax, w=30, vmax=15, pmax=10):
    lstmoyGlouton1 = []
    lstmoyGlouton2 = []
    lstnbOptGlouton1 = []
    lstnbOptGlouton2 = []
    lstabs = list(range(4,nmax+1,max(1,nmax//100)))
    for n in range(4,nmax+1,max(1,nmax//100)):
        moyGlouton1, moyGlouton2, nbOptGlouton1, nbOptGlouton2 = plusieursComparaisons(
        lstmoyGlouton1.append(moyGlouton1)
        lstmoyGlouton2.append(moyGlouton2)
        lstnbOptGlouton1.append(nbOptGlouton1)
        lstnbOptGlouton2.append(nbOptGlouton2)
    plt.title("Pourcentage de succès (l'algorithme trouve la valeur optimale)")
    plt.plot(lstabs,lstnbOptGlouton1)
    plt.plot(lstabs,lstnbOptGlouton2)
    plt.legend(('Glouton1','Glouton2'))
    plt.xlabel("Nombre d'objets")
    plt.show()
    plt.title("Quotient de la valeur renvoyée par l'algorithme glouton sur la valeur op
    plt.plot(lstabs,lstmoyGlouton1)
    plt.plot(lstabs,lstmoyGlouton2)
    plt.xlabel("Nombre d'objets")
    plt.legend(('Glouton1','Glouton2'))
    plt.show()


courbeGloutons(100,100)
```


Pourcentage de succès (l'algorithme trouve la valeur optimale)
Glouton1
Glouton2
100
80
60
40
20
0
0
20
40
60
80
100
Nombre d'objets


Quotient de la valeur renvoyée par l'algorithme glouton sur la valeur optimale
1.0
0.9
0.8
0.7
0.6
0.5
0.4
Glouton1
Glouton2
0
20
40
60
80
100
Nombre d'objets