ENSAE 2011 Examen écrit (1 heure)

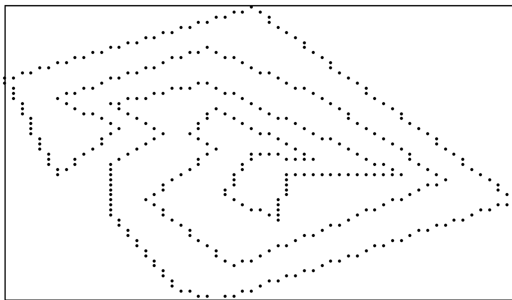
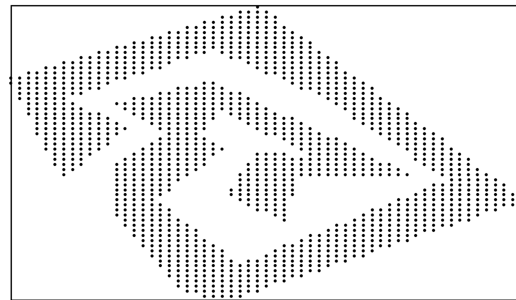
Initiation à la programmation et à l'algorithmique

Xavier Dupré

Tous documents autorisés.

1

On suppose qu'on dispose ici d'une matrice remplie de 0 et de 1. Les cases contenant les valeurs 1 représentent un motif "creux" ou contour qu'on souhaite remplir de 1. Les deux schémas suivants illustrent la matrice avant le remplissage et le résultat qu'on souhaite obtenir après remplissage.

L'algorithme est assez simple : on suppose qu'on connaît une case de l'intérieur du contour. On lui donne la valeur 1. On s'intéresse ensuite à ses quatre voisins (haut, bas, gauche, droite) jusqu'à ce qu'on soit bloqué par le contour. Cela donne la fonction suivante :

```
def remplissage (mat, x , y) :        # ligne 1
    dico = { (x,y):0 }                # ligne 2
    while len (dico) > 0 :            # ligne 3
        point       = dico.popitem ()# retourne un élément du dictionnaire et le supprime
        x,y         = point [0]       # ligne 5
        mat [x][y]   = 1              # ligne 6
        dico [x-1,y] = 0              # ligne 7
        dico [x+1,y] = 0              # ligne 8
        dico [x,y-1] = 0              # ligne 9
        dico [x,y+1] = 0              # ligne 10
```

Malheureusement, le programme ne se termine jamais. Il reste bloqué dans une boucle infinie. Lorsqu'on affiche la taille de `dico` et les variables `x,y` [1], on observe :

```
...
118 41 13
```

1. On utilise l'instruction `print len(dico), x, y`.

```
118 40 13
118 34 8
118 34 7
118 41 13
118 40 13
118 34 8
118 34 7
118 41 13
...
```

1) Expliquez pourquoi le programme n'arrive pas à sortir de la boucle ? (2 points)

2) Corrigez la fonction `remplissage` afin qu'elle permette d'obtenir la seconde figure, à savoir remplir de 1 l'intérieur du contour. (Il n'est pas nécessaire de recopier la fonction, les numéros de lignes serviront à repérer vos modifications.) (3 points)

3) Modifiez la fonction `remplissage` afin qu'elle retourne le nombre de cases modifiées. (2 points)

Correction

1) La séquence de chiffres montre que la boucle est une boucle infinie puisqu'elle reproduit sans cesse le même cycle. A chaque passage dans la boucle, la fonction retire un élément du dictionnaire puis en ajoute au plus quatre. Mais ceci ne veut pas dire obligatoirement que la boucle est une boucle infinie : on ajoute des éléments à un dictionnaire et ils pourraient déjà être présents. Toutefois, comme le montre la sortie du programme, c'est bien une boucle infinie. L'algorithme de remplissage oublie de tester en fait si une case est déjà remplie ou non. Il continue indéfiniment sans se soucier du contour.

2) Il n'existe pas de solution unique, voici la plus courte

```
def remplissage (mat, x , y) :        #
    dico = { (x,y):0 }                #
    while len (dico) > 0 :            #
        point        = dico.popitem ()#
        x,y          = point [0]      #
        if mat [x][y] == 1 : continue # ligne ajoutée
        mat [x][y]   = 1              #
        dico [x-1,y] = 0              #
        dico [x+1,y] = 0              #
        dico [x,y-1] = 0              #
        dico [x,y+1] = 0              #
```

Une autre solution était :

```
def remplissage (mat, x , y) :        #
    dico = { (x,y):0 }                #
    while len (dico) > 0 :            #
        point        = dico.popitem ()#
        x,y          = point [0]      #
        mat [x][y]   = 1              #
        if mat[x-1][y] == 0 : dico [x-1,y] = 0  #
        if mat[x+1][y] == 0 : dico [x+1,y] = 0  #
        if mat[x][y-1] == 0 : dico [x,y-1] = 0  #
        if mat[x][y+1] == 0 : dico [x,y+1] = 0  #
```

3) Pour cette question, il suffisait d'ajouter une variable de comptage.

```
def remplissage (mat, x , y) :
    int compte = 0                      # ligne ajoutée
    dico = { (x,y):0 }
    while len (dico) > 0 :
        point       = dico.popitem ()
        x,y         = point [0]
        if mat [x][y] == 1 : continue
        mat [x][y]   = 1
        dico [x-1,y] = 0
        dico [x+1,y] = 0
        dico [x,y-1] = 0
        dico [x,y+1] = 0
        compte += 1                     # ligne ajoutée
    return comte                        # ligne ajoutée
```

Si on désire un résultat moins précis mais du même ordre de grandeur, il suffisait d'ajouter la ligne `return len(dico)` à la fin.

2

On s'intéresse dans cet exercice au problème du sac-à-dos : il s'agit de remplir le plus possible un sac-à-dos qui ne peut supporter qu'un certain poids avec des objets dont le poids est connu. Par exemple, on dispose d'un sac-à-dos de 15 kilos qu'on doit remplir avec des objets de 2,4,7,10 kilos. La meilleure solution consiste à prendre les objets pesant 4 et 10 kilos.

1)

Le premier algorithme envisagé consiste à parcourir les objets par ordre de poids décroissant. Si un objet peut être ajouté au sac-à-dos sans dépasser la limite alors il est ajouté et on passe à l'objet suivant. Compléter le programme qui suit pour obtenir cet algorithme. (3 points)

```
def resolution_simple (objets, sac) :
    objets.sort (reverse = True)
    solution = []
    for o in objets :
        ...
    return solution
```

2) Que retourne la fonction lorsque le poids du sac est de 10 kilos et qu'il n'y a qu'un seul objet de 12 kilos ? (1 point)

3) Trouver un exemple (une liste d'objets et un poids maximal) pour lequel cet algorithme ne retourne pas la solution optimale. (2 points)

4)

On propose une autre résolution par récurrence. On note la liste des objets $(w_1, ..., w_n)$ et P le poids maximal à ne pas dépasser. On désigne $S(w_1, ..., w_n, P)$ la solution optimale pour le poids P.

$$S(w_1, ..., w_n, P) = \max \{S(w_2, ..., w_n, P), S(w_2, ..., w_n, P - w_1)\} \tag{1}$$

Autrement dit, la solution optimale d'un problème à n objets est la meilleure des deux solutions suivantes :

1. La meilleure solution à $n-1$ objets et le même sac : le premier objet n'est pas sélectionné.

2. La meilleure solution à $n-1$ objets et le sac diminué du poids du premier objet : le premier objet est sélectionné.

On a cherché à implémenter cette solution, on aboutit au programme suivant :

```
def resolution (objets, sac) :
    if len (objets) == 1 :
        if objets [0] <= sac :  return [ objets [0] ]
        else :                  return []

    reduit  = objets [1:]
    s1      = resolution (reduit, sac)
    s2      = resolution (reduit, sac - objets [0])   # ligne B

    t1      = sum(s1)
    t2      = sum(s2) + objets [0]                    # ligne C

    if   sac >= t1 and (t1 >= t2 or t2 > sac) : return s1
    elif sac >= t2 and (t2 >= t1 or t1 > sac) : return [ objets [0], ] + s2

obj = [2,4,7,10]
sac = 15
print "solution ",resolution (obj, sac) # ligne A
```

Le programme déclenche l'exception suivante (les numéros de lignes ont été remplacées par des lettres correspondant aux commentaires insérés ci-dessous) :

```
solution
Traceback (most recent call last):
  File "examen2011.py", line A, in <module>
    print "solution ",resolution (obj, sac)
  File "examen2011.py", line B, in resolution
    s2      = resolution (reduit, sac - objets [0])
  File "examen2011.py", line C, in resolution
    t2      = sum(s2) + objets [0]
TypeError: 'NoneType' object is not iterable
```

Pourquoi ? (2 points)

5) Comment corriger la fonction pour qu'elle retourne le bon résultat ? (3 points)

6) On cherche à estimer le nombre de fois que la fonction `resolution` est appelée. Trouvez un équivalent lorsque n est grand. (2 points)

Correction

1)

```
def resolution_simple (objets, sac) :
    objets.sort (reverse = True)
    solution = []
    for o in objets :
        if sum(solution) + o <= sac :   # ligne ajoutée
            solution.append (o)         # ligne ajoutée
    return solution
```

2) La fonction retourne une liste vide, soit : [].

3) Le résultat devait inclure au moins trois objets et un poids pour le sac. Voici une solution : [10, 9, 2] pour un poids de 11. L'algorithme choisit tout d'abord 10 et s'arrête là car tout autre objet supplémentaire aboutit à un poids supérieur à 11 alors que la solution optimale est 9+2.

4)

Certains élèves ont suggéré qu'il y avait trois erreurs, une pour chaque ligne A,B,C alors qu'il n'y a en fait qu'une seule erreur qui se produit à la ligne C. Les deux autres lignes représentent ce qu'on appelle la pile d'appel : le programme est passé par la ligne A, puis B avant de provoquer une erreur en C.

```
t2      = sum(s2) + objets [0]
TypeError: 'NoneType' object is not iterable
```

Il n'était pas évident de trouver l'erreur, elle pouvait être à trois endroits possibles :

1. `objets`[0] : impossible d'obtenir le premier élément d'un tableau. Peut-être que le tableau est vide ou que `objets` n'est pas un tableau.
2. + : l'addition peut échouer si on cherche à ajouter des éléments qui ne peuvent pas être additionnés.
3. `sum`(s2) : pour une raison quelconque, le programme ne peut pas faire la somme de la liste `s2`.

La première cause est impossible sinon la liste `reduit` = `objets`[1 :] exécutée avant aurait déjà retournée une erreur. La seconde ne l'est pas non plus car le message aurait explicitement mentionnée l'addition. Reste la troisième option : `sum`(s2). Le message stipule qu'un objet est égale à `None`, c'est donc `s2`. Et `None` est n'est pas un tableau ou une liste, il n'est pas itérable et ne peut être utilisé par la fonction `sum`.

Le programme s'arrête donc parce que la variable `s2` contient `None`.

5) Si la variable `s2` contient `None`, cela signifie que la fonction `resolution` a retourné `None` lors de son appel à la ligne B :

```
    s2      = resolution (reduit, sac - objets [0])   # ligne B
```

Comme cette fonction ne contient aucune instruction explicite `return None`, c'est donc que la fonction se termine sans retourner de résultat. Or elle se termine par les deux lignes suivantes :

```
    if   sac >= t1 and (t1 >= t2 or t2 > sac) : return s1
    elif sac >= t2 and (t2 >= t1 or t1 > sac) : return [ objets [0], ] + s2
```

Cela signifie qu'aucune de ces deux conditions n'est vérifiée et que la liste `objets` contient plus d'un élément (voir la début de la fonction). Un autre cas se produit donc et pour le traiter, il suffit d'ajouter :

```
    if   sac >= t1 and (t1 >= t2 or t2 > sac) : return s1
    elif sac >= t2 and (t2 >= t1 or t1 > sac) : return [ objets [0], ] + s2
    else : return []     # ligne ajoutée
```

6) Simplifions l'expression de la fonction `resolution` qu'on appelle f. Soit U_1^n une liste de n objets indicée de 1 à n. P est le poids du sac.

$$f(U_1^n, P) = \begin{cases} f(U_2^n, P) & \text{si ....} \\ f(U_2^n, P - u_1) & \text{si ....} \\ \emptyset & \text{sinon} \end{cases} \tag{2}$$

Cette reformulation fait apparaître une fonction récursive qui s'appelle deux fois dans des conditions différentes. Le coût pour une liste de longueur n suit la suite (c_n) :

$$c_n = c_{n-1} + c_{n-1} = 2c_{n-1} \tag{3}$$

Le coût de la fonction est donc en $O(2^n)$.