4MMAOD6 – TD 4 Programmation dynamique

Préambule : Synthèse du cours

Un problème d'optimisation peut être modélisé par une équation de Bellman lorsque :

1. une solution optimale peut s'exprimer de façon récursive ;
2. la formule récursive réduit la solution du problème général à la résolution optimale d'un ou de plusieurs sous-problèmes de ce problème général (*sous-structure optimale*) ;

L'*équation de Bellman* est une équation récursive qui caractérise la valeur d'une solution optimale sans oublier les conditions initiales. La *programmation dynamique* est une technique pour calculer une solution optimale à partir de cette équation en éliminant les calculs redondants. La méthodologie pour résoudre un problème par programmation dynamique est la suivante :

1. équation de Bellman : prouver la structure récursive d'une solution optimale et en déduire une équation caractérisant la valeur d'une solution optimale de manière récursive ;
2. écrire un programme qui calcule la valeur d'une solution optimale :
   - de manière récursive en utilisant une technique de mémoïsation pour éliminer les appels redondants avec arrêt aux conditions initiales (approche par analyse descendante, dite *top-down*) ;
   - ou de manière itérative en partant des conditions initiales jusqu'au problème à résoudre, les sous-problèmes étant résolus dans l'ordre partiel de leurs dépendances (approche par synthèse ascendante, dite *bottom-up*).
3. mémoriser dans ce programme le choix optimal retenu dans la résolution de chacun des sous-problèmes ; une fois tous ces choix optimaux mémorisés, reconstruire une solution optimale du problème cible de manière descendante (top-down) en listant la séquence de choix optimaux de chaque sous-problème rencontré dans cette solution.
4. analyser les dépendances pour écrire un programme itératif ayant des propriétés de localité mémoire (parcours contigüs en mémoire, blocking).

Rendu de monnaie optimal (durée : 40')

Soit $P = \{p_i, 1 \leq i \leq n\}$ un ensemble de valeurs de pièces avec $p_1 = 1$. On rend un montant S donné avec un nombre minimal $\phi_P(S)$ de pièces. ϕ_P est caractérisé récursivement par : (équation de Bellman)

$$\phi_P(S) = 1 + \min_{k=1}^{n} \{ \phi_P(S - p_k)\} \text{ avec } \begin{cases} \phi_P(0) = 0 \\ \forall S < 0 : \phi_P(S) = +\infty \end{cases}$$

Question 1 Ecrire un programme qui, pour S donné, affiche une liste de $\phi_P(S)$ pièces de P à rendre ; on pourra :

(a) d'abord écrire un programme récursif avec memoïsation qui calcule et stocke dans un tableau les valeurs $\phi(s)$ pour $0 \leq s \leq S$;

(b) dans le déroulé de l'algorithme stocker le type des pièces qui permettent d'obtenir le minimum (tableau `choixOpt[0..S]`) ;

(c) puis écrire un programme qui affiche une liste minimale de pièces à rendre (à partir de `choixOpt[0..S]`)) ;

(d) enfin analyser l'ordre des dépendances (écriture puis lecture) et écrire un programme itératif qui calcule le tableau `choixOpt[0..S]` ;

(e) conclure en analysant les nombres d'opérations et de défauts de cache de votre programme ; ce programme est-il cache-oblivious ?

(a) On dérive le programme directement de l'équation récursive de Bellman :

```
1  int valpOpt[S+1] = {0} ; // valOpt[s] stocke la valeur  \phi(s) une fois calculée (init 0)
2  int choixpOpt[S+1] = {0} ; // choixOpt[s] stocke le type de piece d'un rendu optimal de s.
3
4  int phi( int s )
5  {
6    if ( (s>0) && (valOpt[s] == 0) ) // phi(s) n'a pas déjà été calculé
7    {
8      int min =  phi(s-1) ;  // on rend 1 avec la pièce P[1] == 1
9      int argmin = 1 ; // indice de la pièce à rendre
10     for (int k=2 ; (k <= n) && (P[k] <= s) ; ++k)
11     {
12       int aux = phi( s - P[k] ) ;
13       if  ( aux < min )
14       { min = aux;
15         argmin = k ;
16       }
17     }
18     // Stockage de la valeur $\phi(s)$ et d'un choix optimal correspondant
19     valOpt[s] = min + 1;
20     choixOpt[s] = argmin ;
21   }
22   return valOpt[s] ;
23 }
```

(b) (b) ci-dessus : ajout des lignes 2, 9, 14, 19 (avec choixOpt et argmin)

(c) Ecriture des choix optimaux (top down) :

```
1  cour >> "Liste des pieces à rendres pour un montant " << S << " : "  ;
2  int s = S ;
3  while (s != 0)
4  {
5    indpiecee = choixOptimal[ s ] ;
6    cout << P[ indpiece ] << " "  ;
7    s -= P[ indpiece ] ;
8  }
9  cout << endl :
```

(d) Ordre des dépendances : `choixOptimal[` s `]` est calculé à partir de `choixOptimal[ s- P[k] ]`. D'où le calcul itératif par s croissant :

```
1  int valpOpt[S+1] = {0} ; // valOpt[s] == $\phi$(s)
2  int choixpOpt[S+1] = {0} ; nt phi( int s )
3
4  valOpt[0] = 0 ;
5  for (s = 1; s <= S; ++s )
6  {
7      int min = valOpt[ s-1 ] ;  // on rend 1 avec la piece P[1] == 1
8      int argmin = 1 ; // indice de la pièce à rendre
9      for (int k=2 ; (k <= n) && (P[k] <= s) ; ++k)
10     {
11       int aux = valOpt[ s - P[k] ] ;
12       if  ( aux < min ) { min = aux;  argmin = k ; }
13     }
14     // Stockage de la valeur $\phi(s)$ et d'un choix optimal correspondant
15     valOpt[s] = min + 1 ;
16     choixOpt[s] = argmin ;
17 }
```

(e) Nombre d'opérations : de l'ordre du nombre d'accès en lecture au tableau `valOpt` i.e. $\simeq k \times S$.
Défauts de cache : S/L sur valOpt si on suppose que les éléments `valOpt[`$s - P[n]..s$`]` tiennent en cache (i.e. $Z \gg P[n]$ int). Et sur choixOpt : S/L en écriture et au plus $\phi_P(S)$ en lecture. Le nombre de défauts est environ $2S/L$, si on compte aussi les défauts sur choixOpt (requis pour le calcul) ; comme les S/L défauts sur choixOpt sont requis, le programme est cache-oblivious (à un facteur au plus 2 de l'oprimal).
L'hypothèse $Z \gg P[n]..s]$ tient en cache est raisonnable, et le programme précédent est cache oblivious.
On peut dimunuer l'espace mémoire en remplaçant le tableau valOpt de taille S par un tableau circulaire de taille $\max_{k=1...n} P[k]$.
Remarque : si $Z < \max_{k=1...n} P[k]$, on peut réduire le nombre de défauts de cache par un orodonnacement récursif par blocs.

La roue du million (durée : 30')

Vous participez au jeu télévisé "Stop ou Encore". Ce jeu utilise une roue graduée de 0 à 1000 que l'on peut lancer au maximum 7 fois. Les lancers sont supposés uniformes : chacun des entiers de 0 à 1000 a la même probabilité d'être en haut de la roue lorsqu'elle s'arrête. Après chaque lancer, vous pouvez dire : "'Stop" et vous gagnez alors mille fois le nombre en haut de la roue (donc entre 0 et 1 million d'euros) ; ou "Encore" et vous relancez alors la roue. Après le 7ème et dernier lancer vous n'avez plus de choix et repartez avec mille fois le nombre indiqué par la roue.

Votre stratégie est de maximiser l'espérance de votre gain : vous dites encore "Encore" ssi l'espérance de votre gain en relançant la roue est supérieure au montant actuellement indiqué par la roue.

Question 2 Votre premier lancer est 666, vous relancez et obtenez 751 : dites-vous "Stop" ou "Encore" ?

(a) Quelle est l'espérance de gain lorsque vous lancez la roue la 7ème fois ?

Chaque nombre a la même probabilité $p = \frac{1}{1001}$; l'espérance de gain est $\sum_{i=0}^{1000} \frac{1}{1001} \times i = 500$keuro.

(b) Quelle est l'espérance de gain lorsque vous lancez la roue la 6ème fois ?

L'espérance de gain au 7ème coup étant 500, il y a 2 cas :
— si je tire $i \leq 500$ je relance et l'espérance de mon gain est 500.

— si je tire $i > 500$ je m'arrête et je gagne i.

D'où espérance gain $= \frac{1}{1001}\left(\sum_{i=0}^{500} 500 + \sum_{i=501}^{1000} i\right) = \frac{1}{1001}\left(500 \times 501 + \frac{1000\times1001}{2} - \frac{500\times501}{2}\right) = 500 + 250 \times \frac{501}{1001} \simeq 625,1$.
On intuite que si, au premier des 7 lancers, on tire le nombre de la Bête, 666, on doit dire "Encore".

(c) Soit $g(n)$ l'espérance de gain lorsque vous lancer la roue et qu'il vous reste en tout n lancers. Ecrire l'équation de Bellman associée à votre stratégie en justifiant la sous-structure optimale.

On justifie d'abord l'hypothèse de *sous-structure optimale* : pour maximiser l'espérance de gain $g(n)$, soit je m'arrête, soit je relance auquel cas je joue ensuite pour maximiser l'espérance de gain $g(n-1)$. QED.
Pour $n = 7 \dots 1$ soit X_n la valeur du tirage lorsqu'il en reste en tout n. Les va $X_n \in \{0, 1000\}$ sont iid de loi uniforme. Après X_n il y a 2 cas :

— si $X_n < g(n-1)$ (la probabilité est $\frac{\lceil g(n-1)\rceil}{1001}$) alors on relance et l'espérance de gain est $g(n-1)$;
— si $X_n \geq g(n-1)$, alors on s'arrête et le gain est X_n.

D'où l'équation de Bellman.

$$
\begin{aligned}
g(n) &= \Pr[X_n < g(n-1)] \times g(n-1) + \sum_{i=\lceil g(n-1)\rceil}^{1000} \Pr(X_n = i) \times i \\
&= \frac{\lceil g(n-1)\rceil}{1001} \times g(n-1) + \frac{1}{1001}\sum i = 0^{1000} i - \frac{1}{1001}\sum_{i=0}^{\lceil g(n-1)\rceil - 1} i \\
&= 500 + \frac{\lceil g(n-1)\rceil}{2002}\left(2g(n-1) - \lceil g(n-1)\rceil + 1\right)
\end{aligned}
$$

avec la condition d'arrêt $g(1) = 500$ (ou plus simplement $g(0) = 0$).

(d) Ecrire un programme pour calculer les seuils de décision à chaque coup. Quel est l'espace mémoire requis, le travail et le nombre de défauts de cache sur le modèle CO avec un cache de taille Z chargé par lignes de cache de taille L..

```
# Précalcul des seuils de décision: g est le tableau qui mémoise les décisions
import math
import random
N = 7
g_memo = [0.0] # Initialisation g[0]
for n in range(1, N + 1):
   g_memo += [ 500.0 + math.ceil(g_memo[n-1]) / 2002.0 * ( 2*g_memo[n-1] - math.ceil(
      g_memo[n-1]) + 1.0) ]

# Jeu
for coup in range(1, N+1):
    gain = random.randrange(1001)  # lancer_roue()
    print( gain )
    print ( g_memo[N-coup])
    if (gain > g_memo[N-coup]):
       print( 'STOP' )
       break
    if (coup < N):
       print( 'ENCORE' )
print( 'Mon gain : {}.".format( gain )
```

Espace mémoire : n float ; $W(n) = \Theta(n)$ opérations ; $Q(n, Z, L) = \frac{n}{L}$ défauts de cache.
Le côut amorti par coup est O(1) (optimal) et le nombre de défauts de cache est optimal.

(e) Quelle est l'espérance de gain avant le premier lancer. Que dites-vous après avoir tiré 751 au deuxième lancer ?

Programme xls ou csv `=ARRONDI(500+ARRONDI.SUP(B2;0)/(2*1001)*(2*B2-ARRONDI.SUP(B2;0)+1);2)`

	A	B
1	n	g(n)
2	0	0
3	1	500
4	2	625,12
5	3	695,5
6	4	741,97
7	5	775,36
8	6	800,68
9	7	**820,62**
10	8	836,78
11	9	850,17
12	10	861,46

Au début du jeu l'espérance de mon gain est $g(7) = 820,6$.
L'espérance de gain après le 1er premier lancer (tirage 666) est $g(6) = 800,6$. Je relance et tire 751 au 2ème lancer ; comme $g(5) = 775,3$, je dis "Encore".

Modélisation par équation de Bellman (Durée 10')

Un loueur, qui possède m paires de skis de longueurs respectives $s_1, \ldots, s_m$, doit servir n clients (avec $n < m$) de tailles respectives $t_1, \ldots, t_n$. Il cherche une affectation d'une paire de skis à chaque skieur qui minimise l'écart maximum entre la taille des skis et des skieurs ; autrement dit n indices distincts $i_1, \ldots, i_n$ dans $\{1, \ldots, m\}$ qui minimisent $\max_{k\in\{1,\ldots,n\}} |t_k - s_{i_k}|$.

Question 3 Donner une équation de Bellman qui caractérise une solution optimale.

On a bien une sous-structure optimale : si il reste un sous-ensemble de paires à affecter à un sous-ensemble de skieurs, on peut les affecter optimalement : remarquer ici que c'est suffisant mais non nécessaire. On caratérise l'écart maximum d'une solution optimale. Soit $f(E, k)$ la valeur d'une solution optimale affectant à chacun des k premiers skieurs une paire de skis prise dans $E \subset \{1, \ldots, m\}$. La valeur du coût minimal est $f(\{1, \ldots, m\}, n)$ où

$$f((E, k) = \min_{i\in E} \max\{f(E - \{i\}, k - 1); |t_k - s_i|\}$$

et les conditions initiales : $f(E, 0) = 0\ \forall E \subset \{1, \ldots, m\}$.

L'exercice suivant est donné à titre d'entraînement individuel.

Cageots de fraises (Durée 45')

Rappel (cf cours 4 et TD3 exercice 2). N cageots de fraises doivent être distribués dans M magasins différents. Les politiques de tarification des magasins ne sont pas linéaires, mais le *bénéfice unitaire par cageot* que l'on peut retirer d'un magasin donné dépend du nombre de cageots de fraises distribué dans ce magasin. Les politiques de tarification sont de plus toutes différentes entre les magasinsr iet données : $b_m[n]$ est le bénéfice attendu de la mise en vente de n cageots dans le magasin m. La question est de savoir comment répartir les cageots entre les différents magasins pour **maximiser le bénéfice total** qui s'écrit :

$$B(N,M) = \max_{n_1,\ldots,n_M} \sum_{i=1}^{M} b_i(n_i) \text{ sous la contrainte } \sum_{i=1}^{M} n_i = N$$

Les équations de Bellman s'écrivent :

$$\begin{aligned} B(m,m) &= \max_{k\in\{0,\ldots,n\}} (b_m(k) + B(n-k, m-1)) \quad \forall 1 \le n \le N, \forall 1 \le m \le M, \\ B(n,1) &= b_1(n), \forall 1 \le n \le N \\ B(0,m) &= 0. \end{aligned}$$

en prenant arbitrairement $b_i(0) = 0$ pour tout i.

On rappelle le cœur du calcul itératif d'une répartition optimale de N cageots de fraises dans M magasins (cf cours et annexe ci-après).

```
int choixOpt[M][N+1] ; // Tableau des choix optimaux
double Benef[N+1] ; // Benefice optimal (en place, par ligne)
for (int p=0 ; p <=N; ++p )  {  // Init
   Benef[p] = b[0][p];  choixOpt[0][p] = p;
}
for (int m=1 ; m <M; ++m ) {
   for (int p=N ; p >= 0 ; --p )  {
      int max = b[m][p] ;
      int argmax = p ;
      for (int k=0 ; k <= p-1 ; ++k )  {
         double tmp =  b[m][k] + Benef[p-k] ;
         if ( tmp > max ) { max = tmp ; argmax = k ; }
      }
      Benef[p] = max ;
      choixOpt[m][p] = argmax ;
   }
}
```

Question 4 Compléter l'algorithme itératif pour écrire une solution optimale à l'écran (i.e. le nombre de paniers attribués à chaque magasin). Donner le travail en nombre de comparaisons, le coût en temps et en mémoire de toute l'algorithme.

Le programme parcourt m cases de la table des choix optimaux, qui en possède $(n+1)(m+1)$. Remarque : A priori un programme récursif prendrait $O(m)$ en mémoire, sauf si le compilateur élimine l'appel récursif terminal (trivial ici).

```
    cageotsIteratif
    solution = []
    for i in range(0, m):
        solution += [0]
    nbpanierrestant = n
    for i in xrange(m-1, -1, -1) :
        solution[i] = choixOpt[i][nbpanierrestant]
        nbpanierrestant = nbpanierrestant - solution[i]
    return {"solution": solution, "bénéfice": tab[n]}
```

Le travail est $n.m$ comparaisons pour le calcul de choixOpt, 0 pour la remontée de la solution. Le temps est $\Theta(nm) + \Theta(m) = \Theta(nm)$. Le coût mémoire est $(n+1)(m+1)$: $(n+1).(m-1)$ pour choixOpt et $(n+1)$ pour tab (solution peut etre stocké en place dans tab). NB Une fois le calcul de la solution terminé, on peut libérer ces 2 tables.

Question 5 On considère le modèle de cache CO avec un cache de taille Z chargé par lignes de cache de taille L. On suppose que l'exécution de la boucle interne `for p=N .. 0` tient en cache (ie $Z \gg 2N$). Analyser le nombre de défauts de cache du porgramme sur chacun des 3 tableaux : b, Benef, choixOpt.

— Benef : une fois Benef (de taille N) en cache, il n'en sort pas ; donc N/L défauts sur Benef ;

— b : est parcouru par ligne : donc NP/L défauts sur b ;

— choixOpt : est parcouru par ligne : donc NP/L défauts sur choixOpt.

Le programme est optimal en nombre de défauts de cache si $2N \ll Z$.

Question 6 Dans cette question uniquement, on indexe les tableaux en intervertissant lignes et colonnes : `b[n][m]` stocke le bénéfice attendu de la mise en vente de n cageots dans le magasin m (de même pour `choixOpt[n][m]`). Que deviennent les nombres de défauts si $2N \ll Z \ll 4N$?

Les tableaux ChoixOpt et Benef sont alors parcourus par colonnes engendrant donc NM défauts (1 par élément) chacun en pire cas : c'est bien moins performant !

Question 7 On suppose maintenant que $N \ll Z$: même le tableau Benef ne tient pas en cache. Analyser le nombre de défauts.

La boucle interne sur k engendre des défauts de cache capacitifs sur b et Benef, pas sur choixOpt. $Q(N, M, L, Z) == \frac{M(N+1)}{L} + 2\frac{MN^2}{2L} = \Theta\left(\frac{MN^2}{L}\right)$.

Question 8 On suppose toujours que $N \ll Z$. On propose de calculer le tableau bidimensionnel $M \times N$ ChoixOpt par un algorithme qui découpe les tableaux en $\frac{M}{K_M} \times \frac{N}{K_N}$ tableaux de taille $K_M \times K_N$.

1. Justifier que cela est possible en précisant les valeurs à stocker.
2. Préciser l'ordre des calculs de blocs en précisant quels blocs peuvent être calculés en parallèle.
3. Analyser le nombre défauts de cache en supposant K_N et K_M tels que 4 blocs $K_M \times K_N$ tiennent en cache.
4. Comment choisiss-vous K_N et K_M ?
5. Que donnerait une version récursive (cache oblivious) ?

1. Pour faire le calcul, on utilise 3 tableaux $M \times (P+1)$: b, ChoixOpt et Benef (qui ici est aussi bidim !). Le coût mémoire est donc $3M(P+1)$ au lieu de $2M(P+1)$ dans la version précédente. Ce léger surcout mémoire permet de rendre le calcul parallèle en restant local au niveau de chaque bloc.
2. Un bloc d'indice (i, j) peut être calculé dès que tous les blocs (k, l) avec $i - 1 \leq i' \leq i$ et $0 \leq j' \leq j$ ont été calculés (seule la dernière ligne est utilisée des blocs de niveau $i - 1$, sur les magasins). Ainsi les blocs situés sur une même diagonale peuvent être faits en parallèle. On traite les diagonales séquentiellement l'une après l'autre.
3. L'opération de base manipule 4 blocs : 2 blocs de Benef, 1 bloc choixOpt, 1 bloc b. En supposant que ces 4 blocs tiennent en cache (avec $4K$ lignes de cache en plus pour le passage au bloc suivant), chaque mise à jour d'un bloc prend $4\frac{K^2}{L}$ défauts de cache. Le nombre de défauts de cache est donc $\sum_{i=1}^{M/K} \sum_{j=1}^{P/K} 2 \times j \times \Theta\left(\frac{K^2}{L}\right) = \Theta\left(\frac{MP^2}{KL}\right) = \Theta\left(\frac{MP^2}{\sqrt{Z}L}\right)$.
   On a donc un nombre de défauts de cache inférieur à la version par blocs précédente, et on gagne en parallélisme pour le prix d'une augmentation de 50% de l'espace mémoire (3 matrices $M \times P$ au lieu de 2). De plus la parallélisation traite efficacement les blocs de taille $K \times K$, sans communication, localement sur chaque cœur (dès que le bloc rentre dans le cache du cœur).
4. On peut faire une version cache oblivious en découpant sur la plus grande dimension jusqu'à un seuil S. On obtient un programme parallèle avec un parallélisme récursif facile à exploiter (en OpenMP par exemple) et qui fait un nombre de défauts de cache $O\left(\frac{MP^2}{L\sqrt{Z}}\right)$ à chaque niveau (L, Z) de la hiérarchie mémoire.

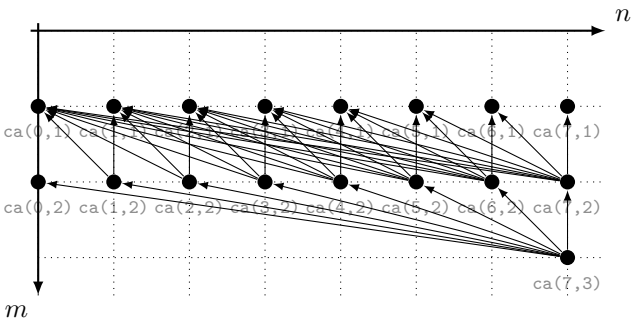
Annexe : code python pour les cageots de fraises

L'équation de Bellman conduit directement au programme récursif avec memoïsation suivant (cf cours et TD3) en Python : l'entrée $b_m(n)$ est stockée dans la cellule $b[m-1][n]$ du tableau b.

```
def cageots(b, N, M):  # appel principal
    """
    b est la fonction bénéfice (définie comme une liste imbriquée)
    N est le nombre de cageots,
    M est le nombre de magasins (de 1 à M)
    """
    memo = []
    choixOpt = []
    for i in xrange(0, m + 1): # Allocation et initialisation memo et choixOpt
        memo += [[]]
        choixOpt += [[]]
        for j in xrange(0, n + 1):
            memo[i] += [-1]
            choixOpt[i] += [-1]
    return cageotsceRecMemoAux(b, n, m, memo, choixOpt)

def cageotsRecMemo(b, n, m, memo, choixOpt):
    if memo[m][n] == -1: # sinon déjà calculé et mémorisé dans memo
        if (n == 0):
            memo[m][n] = 0
            choixOpt[m][n] = 0
        elif (m == 1): # On met les n cageaots dans le magasin 1
            memo[m][n] = b[0][n] # Précondition: b[0][n] est croissant avec n
            choixOpt[m][n] = n
        else:
            valopt = b[m-1][n]  # n cageots dans le magasin m
            choixvalopt = n
            for i in xrange(0, n): # i cageaots dans le magasin m
                tmp = b[m - 1][i] + cageotsRecMemo(b, n - i, m - 1, memo, choixOpt)
                if ( tmp > valopt ):
                   valopt = tmp
                   choixvalopt = i
            memo[m][n] = valopt
            choixOpt[m][n] = choixvalopt
    return memo[m][n]
```

Les dépendances des appels (i.e. sur `memo[n][b]`) sont décrites dans le graphe ci-dessous qui a deux axes : k allant de 0 à n et i allant de 1 à m. Chaque $B(k,i)$ d'une ligne (ou colonne) i donnée dépend de tous les éléments de la ligne précédente $i-1$ de 0 à k. Le graphe de dépendance des appels (ci-dessus) montre que, pour un i donné, les $B(k,i-1)$ suffisent pour calculer tous les $B(k,i)$. Le



tri topologique va donc parcourir les nœuds ligne i par ligne i. On a deux possibilités : soit on parcourt selon les k croissants, soit on parcourt selon les k décroissants. Dans le premier cas, on aura besoin, à tout moment, de stocker deux lignes. Dans le second cas, une ligne suffit car un certain $B(k,i-1)$ ne sert pas pour calculer les $B(j,i)$ pour $j<k$. On peut donc remplacer $B(k,i-1)$ par $B(k,i)$ quand celui-ci est calculé.
Le tri topologique considéré ici est donc : $(n,1) < (n-1,1) < \cdots < (0,1) < (n,2) < \cdots < (0,2) < \cdots < \cdots < (n,m) < \cdots < (0,m)$ et conduit à l'algorithme itératif suivant.

```
def cageotsIteratif(b, n, m):
    """
    b est la fonction bénéfice (définie comme une liste imbriquée)
```

```
    n est le nombre de cageots,
    m est le nombre de magasins
    """
    tab = []
    choixOpt = [[]]
    tab += [0]
    choixOpt[0] += [0]
    for k in range(1, n + 1):
        choixOpt[0] += [k]
        tab += [b[0][k - 1]]


    for l in range(1, m - 1):
        # Calcul de choisOpt[l][i] pour 0<= i <= n
        choixOpt += [[]]
        for k in range(0, n + 1):
            choixOpt[l] += [0]
        for k in range(n, -1, -1):
            valOpt = tab[k] # 0 cageots dans magasin l
            solOpt = k
            for i in range(1, k + 1):
                tmp = b[l][i - 1] + tab[k - i]
                if (tmp > valOpt):
                    valOpt = tmp
                    solOpt = i
            tab[k] = valOpt
            choixOpt[l][k] = solOpt

    # Optimisation : pour le magasin m, seul choixOpt[m-1][n] est calcule
    tab[n] = tab[n]
    choixOpt += [[]]
    for k in range(0, n + 1):
        choixOpt[m - 1] += [0]
    for i in range(1, n + 1):
        tmp = b[m - 1][i -  1] + tab[n - i]
        if (tmp > tab[n]):
            tab[n] = tmp
            choixOpt[m - 1][n] = i
```