TD 3 d'Algorithmique et Optimisation Discrète

Mémoïsation et programmation dynamique

Équipe pédagogique AOD

Résumé du cours : programmes récursifs, mémoïsation et blocking

La caractérisation récursive d'un problème peut conduire à des calculs redondants, si une fonction pure est appelée plusieurs fois avec les mêmes paramètres. Pour les éliminer, la méthodologie est la suivante :

- identification des calculs redondants en dessinant le graphe des appels : chaque sommet v représente un appel de la fonction pour une valeur fixée des paramètres ; il y a un arc de u à v ssi lors de l'exécution de u, on appelle directement v.
- élimination des calculs redondants, avec deux techniques :
  1. mémoïsation, c'est à dire stockage du résultat lors du premier appel, puis utilisation directe de ce résultat lors des appels ultérieurs :
     - soit avec un opérateur fonctionnel de mémoïsation (qui utilise une table de hachage pour le stockage des appels avec comme clef les paramètres de l'appel) ;
     - soit à la main en modifiant le code récursif pour introduire un tableau qui stocke le résultat de chaque appel.
  2. ordonnancement des calculs selon un ordre topologique des nœuds du graphe d'appels (à partir des puits) : l'avantage est d'économiser la place mémoire. Ce procédé conduit à des algorithmes itératifs (construction ascendante ou "bottom-up"), généralement plus efficaces en nombre d'opérations (i.e. travail) que la mémoïsation du programme récursif.

Pour prendre en compte la localité et diminier les défauts de cache, on privilégie des ordonnancements bloc-itératifs (cache-aware) ou bloc-récursifs (cache-oblivious) grâce à la technique de *blocking* : regroupement des opérations en bloc de calcul de grande taille s'exécutant en cache sans défaut autre que les obligatoires. Cette technique est aussi utile pour mettre en évidence le parallèlisme potentiel entre instructions indépendantes.

Application : caractérisation récursive de la valeur d'une solution optimale. On appelle *optimisation discrète* la recherche d'une solution optimale dans un ensemble fini. Il est parfois possible de caractériser de manière récursive la solution optimale d'un problème (à partir des valeurs optimales de sous-problèmes). Cette caractérisation conduit naturellement à un programme récursif inefficace : les techniques de memoïsation (pour éliminer la redondance) –on parle de **programmation dynamique** en référence à la tabulation des valeurs, comme dans un tableur– et de blocking (pour augmenter la localité et mettre en évidence le parallélisme) construisent un programme plus efficace.

1 Rendu de monnaie optimal (Durée attendue : 15')

Une machine rend la monnaie avec les pièces suivantes : $1c, 2c, 5c, 10c, 20c, 50c, 100c, 200c$. Toutes les pièces sont supposées en nombre suffisant. Pour toute somme s à rendre, la machine rend un nombre minimal $\phi(s)$ de pièces.

Question 1 Quelle est la valeur de $\phi(s)$ pour $s = 0, \ldots, 10$?

$\phi(0 = 0; \phi(1, 2, 5, 10) = 1; \phi(3, 4, 6, 7) = 2; \phi(8, 9) = 3.$

Question 2 Plus généralement, soit $P = \{p_i, 1 \leq i \leq n\}$ un ensemble de valeurs de pièces (en question 1, $P = \{1, 2, 5, 10, 20, 50, 100, 200\}$.)
Donner une équation récursive (dite de Bellman) caractérisant $\phi_P(s)$ pour le problème général.

$\phi_P(s) = 1 + \min_{k=1}^{n} \{ \phi_P(s - p_k)\}$ avec les conditions d'arrêt : $\phi_P(0) = 0$s et pour tout $s < 0 : \phi_P(s) = +\infty$.
NB Dans le cas où l'algorithme glouton est optimal (ie pour les systèmes monétaires dits *canoniques* : $x_n = \left\lfloor \frac{s}{p_n} \right\rfloor$ et $x_k = \left\lfloor \frac{s - \sum i=k+1^n x_i p_i}{p_k} \right\rfloor$ ou encore $\phi(s) = 1 + \phi\left(s - \max\{p_k \in P, p_k \leq s\ \}\right)$.

Remarque. Le problème général se modèlise par un programme linéaire en nombre entiers (PLNE) :
trouver $x \in \mathbb{N}^n$ qui minimise $\sum_{i=1}^{n} x_i$ sous la contrainte $\sum_{i=1}^{n} x_i \times p_i = s$.
Ce problème est NP-difficile ; dans le cas général, on ne connait pas à ce jour d'algorithme polynomial en la taille de l'entrée s, i.e. qui fait $\log^{O(1)} s$ opérations.
L'algorithme glouton classiquement utilisé – *Tant qu'il reste quelque chose à rendre, choisir la plus grosse pièce qu'on peut rendre*– fait $\tilde{O}(\log s)$ opérations donc est rapide, de temps quasi linéaire en la taille de l'entrée s ; mais il ne rend pas un nombre minimal de pièces en général sauf pour les systèmes monétaires dits *canoniques*, comme par exemple le cas particulier de l'énoncé où les valeurs des pièces sont *super croissantes* i.e. $\forall i > 1 : p_i \geq \sum_{k=1}^{i-1} p_k$.
Avec $P = \{1, 2, 4, 5\}$, l'algorithme glouton rend 3 pièces pour $s = 8$ alors que l'optimal est 2 pièces de 4.

2 Coefficients binomiaux $\binom{n}{p}$ (Durée attendue : 50')

Les coefficients binomiaux $C_n^p = \binom{n}{p}$ sont définis par la récurrence (triangle de Pascal) :

$$\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1} \quad \text{pour } 0 < p < n; \qquad \text{et } \binom{n}{0} = \binom{n}{n} = 1.$$

Soit $\mathcal{T}_{n,p}$ l'ensemble des couples (i, j) tels que $\binom{i}{j}$ est utilisé avec cette formule pour calculer $\binom{n}{p}$.

Cette formule conduit au programme récusrsif suivant (sans mémoïsation) :

```
1 def binom(n,p):
2     if (p==0) or (p==n) :
3         return 1
4     else :
5         return binom(n-1,p) + binom(n-1, p-1)
```

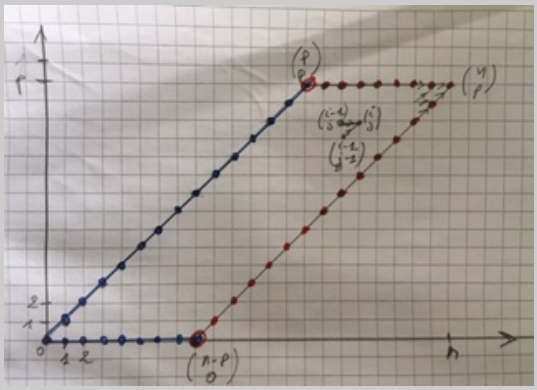
Ce programme récursif naïf effectue [1] $W_+(n, p) = \binom{n}{p} - 1$ opérations d'addition et effectue donc $2\binom{n}{p} - 1$ appels ; le coût est exponentiel en pire cas [2].

Question 3 Dessiner le graphe des appels et montrer qu'il y a des appels redondants. Combien d'appels sont utiles (non redondants) ?

1. L'arbre des appels récursifs est un arbre binaire localement complet, de racine l'appel `binom(n,p)` ; les feuilles sont les appels qui retournent la valeur 1, il y en a donc $\binom{n}{p}$. Or un arbre binaire à m feuilles possède $m-1$ nœuds interne ; il y a donc $\binom{n}{p} - 1$ nœuds internes, chacun correspondant à une addition.

2. En effet $W_+(n, p) = \frac{n!}{(p!(n-p)!} = \prod_{i=1}^{p} \frac{n-p+i}{i}$. D'où pour $n = 2p$: $W_+(2p, p) = \prod_{i=1}^{p} \frac{p+i}{i} \geq 2^p$ (car $i \leq p$ donc $p + i \geq 2i$). Le nombre d'additions en pire cas est au moins exponentiel.

Il y a (au moins) 2 appels avec $(n-2, p-1)$: une fois à partir de $(n-1, p)$ et une fois à partir de $(n-1, p-1)$. Les appels utiles sont dans le triangle de Pascal ci-dessous, qui compte $p(n-p)$ appels utiles (et en plus les $p+(n-p)$ appels $\binom{k}{k}$ et $\binom{k}{0}$) :



On peut aussi compter les points : $(p+1)(n-p+1)-1$: $p+1$ lignes de $n-p+1$ colonnes, mais il faut enlever le point $(0,0)$ qui n'est jamais appelé, et on retombe sur $p(n-p)+n$. Du coup, (0,0) est un cas particulier : 1 apel utile (obligatoire).

Question 4 Ecrire un programme récursif avec mémoisation et analyser : la place mémoire allouée ; le coût amorti par coefficient de $\mathcal{T}_{n,p}$ en nombre d'additions ; le nombre total d'opérations (notation O ou Θ).

```
def binom_memo(n,p) :
    mem  = {}
    for i in range(n+1):
        for j in range (p+1):
            mem[(i,j)] = -1
    def binom_memo_rec(n,p):
        if (mem[(n,p)] == -1) :
            if (p==0) or (p==n) :
                mem[(n,p)] = 1
            else :
                mem[(n,p)] = binom_memo_rec(n-1,p) + binom_memo_rec(n-1,p-1)
        return mem[(n,p)]
    return binom_memo_rec(n,p)
```

La place mémoire allouée est ici $(p+1)(n+1)$. Le coût amorti est 1 addition par coefficient, soit $O(1)$. Pour l'appel `binom_memo(n,p, mem)`, le nombre d'additions est donc $p(n-p)$ d'où un coût $\Theta((n-p)p)$ Mais l'initialisation du tableau de taille np à -1 demande un coût $O(np)$.
Remarque : on pourrait se limiter à allouer un tableau T de taille $(n-p)\times p$: il suffit pour cela de stocker $\binom{i}{j}$ dans $T[i-j][j]$. Le coût total est alors bien $\Theta(p(n-p))$.

Question 5 Ecrire un programme itératif sans calcul redondant qui calcule et stocke tous les coefficients binomiaux utillisés dans le calcul de $\binom{n}{p}$; préciser le nombre d'opérations (par exemple additions) et la place mémoire requise.

```
def binomial_iter(n, p):
    # On stocke binom(i,j) dans la case T[i-j][j]
    T  = [[1] * (p+1) for _ in range(n-p+1)]
    for j in xrange(1, p + 1):
        for i in xrange(1, n-p + 1):
            T[i][j] = T[i][j-1] + T[i-1][j]
    return T[n-p][p]
```

Le nombre d'additions est $(n-p)p$ d'où le coût $\Theta((n-p)p)$. La place mémoire allouée est ici $(p+1)(n-p+1)$.

Question 6 On ne désire plus stocker tous les $\binom{i}{j}$ en mémoire mais juste les calculer et les afficher à l'écran ; comment réduire la mémoire ? Analyser le nombre de défauts de cache sur le modèle CO (cache de taille Z chargé par blocs de taille L avec politique d'écrasementLRU).

; on peut améliorer en place mémoire $\min(p, n-p)+1$ en faisant un calcul en place, par ligne ou par colonne (ou diagonale) en mettant en boucle interne la plus petite dimension entre p et $n-p$: :

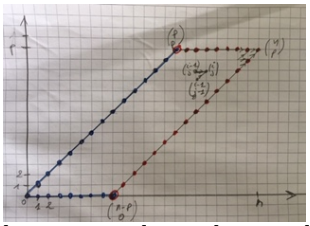
```
def binomial_iter_enplace(n, p):
    # calcul avec en boucle interne la plus petite dimension
    if (p > n-p ) : dim1, dim2 = p, n-p
    else          : dim1, dim2 = n-p, p
    T  = [1] * (dim2+1)
    for j in xrange(1, dim1 + 1):
        for i in xrange(1, dim2 + 1):
            T[i] = T[i] + T[i-1]
    return T[n-p]
```

En C le programme s'écrit en suivant les dépendances ; on suppose ici $p > n-p$.

```
Integer binom( int n, int p )
{  Integer *L = (Integer*) malloc( (n-p+1)*sizeof(Integer) )  ; // init: ligne p=0
   for (int k=0; k <= n-p; ++k) L[k] = 1; // Init
   /* A l'entrée  de la boucle j: L[k] stocke C(k+j-1, j-1) pour k=0 .. n-p
    * A la sortie de la boucle j: L[k] stocke C(k+j, j) pour k=0 .. n-p
    */
   for (int j=1; j <= p; ++j)
      for (int i=1; i <= n-p; ++i) L[i] = L[i-1] + L[i] ;
   Integer res = L[n-p] ;
   free ( L ) ;
   return res ;
}
```

- espace mémoire : $S(n,) = n - p = \min(p, n-p)$: il est minimal.
- nombre d'additions : $W_+(n,p) = p(n-p)$: il est minimal (par rapport à la formule récursive).
- localité :
  - si le programme s'exécute en cache : $Q(n,p,L,Z) = \frac{n-p+1}{L} + \frac{p+1}{L} + O(1) = \frac{n}{L} + O(1)$.
  - si le cache est très petit (i.e. $Z \ll n-p$) : $Q(n,p,L,Z) = p \times \left(\frac{n-p+1}{L} + O(1)\right) = \frac{(n-p)p}{L} + O(p)$. Ce n'est a priori pas optimal.

Question 7 Pour améliorer la localité, on applique la technique de blocking.
On partitionne les W_+ calculs en blocs tel qu'un bloc de calcul s'exécute sans défaut de cache (une fois les données nécessaires au bloc chargées en cache) et maximise le nombre d'opérations effectuées avec un cache de taille Z.
Pour cela, on partitionne le graphe de dépendance en parallélogrammes comportant $K_1 \times K_2$ additions à effectuer, dont les entrées sont les K_1 coefficients $\binom{I+k}{J}$ pour $0 \le k \le K_1$) et les K_2 coefficients $\binom{I+k}{J+k}$ pour $0 \le k \le K_2$).



Le programme suivant implémente le calcul d'un bloc en place dans deux tableaux préalloués $L[0..n-p+1]$ et $D[0..p+1]$:

```
Integer* L=0 ; // Assumed preallocated and intialized
Integer* D=0 ; // Assumed preallocated and intialized

void calculBloc(int I, int J, int K1, int K2)  // Precondition: 0 <= J <= I
   /* A l'entrée de ce bloc:
    *    L[I+k] stocke C(I+k, J) pour 0 <= k <= K1
    *    D[J+k] stocke C(I+k, J+k) pour 0 <= k <= K2
    * A la sortie du bloc:
    *    L[I+k] stocke C(I+k, J+K2-1) pour 0 <= k <= K1
    *    D[J+k] stocke C(I+K1-1+k, J+K1-1+k) pour 0 <= k <= K2
    */
{ for (int j=J+1; j <= J+K2 ; ++j)
   {  D[j-1] = L[I+K1] ;
      L[I] = D[j];
      for (int i=I+1; i <= I+K1; ++i) L[i] = L[i-1]+L[i] ; // acces contigus
   }
   D[J+K2] = L[I+K1];
}
```

On suppose que le cache de taille Z contient les $(K_1+1)+(K_2+1)$ éléments : $L[i]$ et D_j pour $I \le i \le I+K_1$ et $J \le j \le J+K_2$.
Montrer que le choix de $K_1 = K_2 \simeq \frac{Z}{2}$ maximise le nombre d'additions sans défauts de cache.

On a $K_1 + K_2 + O(1) \simeq Z$ et le bloc effectue $K_1 \times K_2$ additions. Le produit de deux nombres positifs de somme constante est maximum lorsque les deux nombres sont égaux. Donc $K_1 = K_2 \simeq Z/2$.

Question 8 En déduire un programme cache aware. Analysez son nombre de défauts de cache et son surcoût arithmétique par rapport au programme de la question 6.

On fait le calcul bloc par bloc, par exemple par ligne (ou par antidiagonale pour exhiber le parallélisme entre blocs).

```
Integer  binom_cacheaware (int n, int p, int K) {
   //cache aware block size:  K=Z/2 - epsilon ;

   L = (Integer*) malloc( (n-p+1) * sizeof(Integer) ) ; // Allocate and intialize L
   for (int  i=0; i<=n-p; ++i) L[i] = ((Integer)(1)) ;
   D = (Integer*) malloc( (p+1) * sizeof(Integer) ) ; // Allocate and intialize D
   for (int  j=0; j<=p; ++j) D[j] = ((Integer)(1)) ; // Init diagonale

   for (int I=0; I <= n-p; I += K) {
      int K1 = ( (I+K <= n-p) ? K : n-p-I ) ;
      for (int J=0; J <= p; J += K) {
         int K2 = ( (J+K <= p) ? K : p-J ) ;
         calculBloc ( I, J, K1, K2 ) ;
   }  }
   Integer res = D[p] ;
   free (L); L=0; // Deallocate L
   free (D); D=0; // Deallocate D
   return res;
}
```

— Nombre de défauts de cache : $Q(n,p,L,Z) = \frac{n-p}{K}\frac{p}{K}\left(2\frac{K}{L} + O(1)\right) = \frac{(n-p)p}{ZL}\left(2 + O\left(\frac{L}{Z}\right)\right) = O\left(\frac{(n-p)p}{ZL}\right)$.

— Surcoût arithmétique : la génération des blocs, soit $O\left(\frac{np}{K^2}\right) = O\left(\frac{np}{Z^2}\right)$.

Question 9 Décrire le principe d'un algorithme cache oblivious et analyser son espace mémoire et sa performance sur une hiérarchie mémoire.

Principe : on découpe récursivement la dimension la plus grande jusqu'à un seuil S.

— L'espace mmoire requiis est $n - p + p = n$.

— Le nombre de défauts de cache au niveau i de la hiérarche avec un cache LRU de taille Z_i chargé par ligne de cache de taille L_i est comme pour le cache aware $O\left(\frac{(n-p)p}{Z_i L_i}\right)$.

— Le surcoût de récursivité est en $O\left(\frac{p(n-p)}{S^2}\right)$. En prenant S de l'ordre de 10 ou 20 il devrait être inférieur à 5%.

Le code (non demandé) sécrit simplement avec la procedure calculBloc :

```
static int S ;
void calculBlocRec( int I, int J,  int K1, int K2 )
{  if ((K1 < S) && (K2 < S) ) calculBloc( I, J, K1, K2) ;
   else
   {  if ( K1 > K2 )
      { calculBlocRec( I, J, K1/2, K2) ;
        calculBlocRec( I+K1/2 , J, K1 - K1/2, K2) ;
      }
      else
      { calculBlocRec( I, J, K1, K2/2) ;
        calculBlocRec( I, J+K2/2, K1, K2-K2/2) ;
}  }  }

Integer  binom_CO (int n, int p, int Seuil)
{  Integer res;
   S = Seuil ;
   // Appel principal
   L = (Integer*) malloc( (n-p+1) * sizeof(Integer) ) ;
   for (int  i=0; i<=n-p; ++i) L[i] = ((Integer)(1)) ; // Init ligne
   D = (Integer*) malloc( (p+1) * sizeof(Integer) ) ;
   for (int  j=0; j<=p; ++j) D[j] = ((Integer)(1)) ; // Init diagonale
   calculBlocRec(0,0, n-p, p )  ;
   res = D[p] ;
   free (L) ;
```

```
    free (D) ;
    return res ;
}
```

Les 2 appels récursifs dans `calculBlocRec` sont séquentiels (dépendance de données entre la sortie du premier appel et l'entrée du deuxième) ; la variante `calculBlocRecPar` ci-dessous permet de mettre en évidence du parallélisme dans les appels récursifs.

```
void calculBlocRecPar( int I, int J,  int K1, int K2 )
{  if ((K1 > S) && (K2 > S) )  // Parallelism available
   {    calculBlocRecPar( I, J, K1/2, K2/2) ;
        calculBlocRecPar( I+K1/2, J, K1-K1/2, K2/2) ; // Both calls are
        calculBlocRecPar( I, J+K2/2, K1/2, K2-K2/2) ; // parallel !
        calculBlocRecPar( I+K1/2, J+K2/2, K1-K1/2, K2-K2/2) ;
   } else if (K1 > S) // here K2 <= S
   {    calculBlocRecPar( I, J, K1/2, K2) ;
        calculBlocRecPar( I+K1/2 , J, K1 - K1/2, K2) ;
   } else if (K2 > S) // here K1 <= S
   {    calculBlocRecPar( I, J, K1, K2/2) ;
        calculBlocRecPar( I, J+K2/2, K1, K2-K2/2) ;
   }
   else calculBloc(I, J, K1, K2) ;
}
```

Ci-dessous, mesures de temps et des accès et défauts de pages des programmes C mesurés avec getrusage.

$n = 200000, p = 100000$	Elapsed Time (s)	CPU Time(s)	System(s)	#Page access	#Page faults
Q2 Calcul itératif	26.3538	26.2993	0.05448	197	0
Q5 Cache aware (K=64)	26.6965	26.6441	0.052411	196	0
Q6 Cache oblivious (S=64)	27.1182	27.0646	0.053665	0	0
Q6 Cache oblivious parallèle (S=64)	27.0035	26.9544	0.0491	0	0

Question 10 Justifier que le seul coefficient $\binom{n}{p}$ est calculé avec $O(\min\{p, n-p\})$ multiplications sur des entiers exacts. Comparer au coût amorti par coefficient des programmes ci-dessus.

```
def binomial_direct(n, p):
    if p > n: return 0
    if p == n: return 1
    k= min(p, n-p)
    num =  1
    for i in xrange(n-k+1, n+1): num *= i  # note n-k == max(p, n-p)
    denom = 1
    for i in xrange(1, k+1): denom *= i
    return num / denom
```

La place mémoire allouée est ici $O(1)$; Le nombre de multiplications est $2\min\{n-p, p\}$. Ce coût est plus intéressant que le coût $\Theta(np)$ des programmes précédents si on ne veut que C_n^p ; mais il est très supérieur au coût amorti de 1 addition (optimal !) si on doit calculer tous les coefficients.

3 Cageots de fraises (Durée attendue : 15')

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 magasins. La question est de savoir comment répartir les cageots entre les différents magasins pour **maximiser le bénéfice total**.

En entrée, on connait $b_i(n_i)$ le bénéfice total (pas unitaire) dans le magasin i obtenu pour la distribution de n_i cageots dans ce magasin, avec i dans $\{1, \ldots, m\}$, n_i dans $\{1, \ldots, n\}$ et $b_i(n_i)$ un entier positif (dans $\mathbb{N}$).

Le bénéfice maximal $B(n, m)$ parmi toutes les distributions possibles des cageots dans les magasins s'écrit :

$$B(n,m) = \max_{n_1,\ldots,n_m} \sum_{i=1}^{m} b_i(n_i) \text{ sous la contrainte } \sum_{i=1}^{m} n_i = n$$

Exemple de tarification pour $m = 3$ magasins, avec des exemples de solution optimale (c'est-à-dire une distribution des cageots dans les magasins et le bénéfice rapporté) dans le cas où il y a :

— $n = 3$ cageots : distribution optimale $(0, 2, 1)$, bénéfice 17 ;
— $n = 5$ cageots : distribution optimale $(0, 2, 3)$, bénéfice 27 ;
— $n = 7$ cageots : distribution optimale $(7, 0, 0)$, bénéfice 45.

On voit que la distribution change complètement : ceci est du au fait que les *gains marginaux* (bénéfice apporté par le $j + 1$-ème cageot relativement au bénéfice apporté par les j cageots précédents : $g_i(j) = b_i(j) - b_i(j-1)$) évoluent de manière non linéaire.

n_i	$b_1(n_1)$	$b_2(n_2)$	$b_3(n_3)$
0	0	0	0
1	3	6	5
2	7	12	10
3	12	16	15
4	17	20	20
5	26	22	25
6	35	24	30
7	45	26	35

Question 11 Justifier la formulation récursive : $B(n, m) = \max_{k=0..n}\{b_m(k) + B(n-k, m-1)\}$. Quelles sont les conditions initiales ?

Soit k le nombre de cageots du magasin m dans une répartition optimale sur les m premiers magasins. Alors nécéssairement les $n - k$ cageots restants sont répartis optimalement dans les $m - 1$ magasins restants.
Condition initiale : $B(0, i) = 0$ pour tout $i \leq m$ car il n'y a pas de bénéfice sans cageot. $B(j, 0) = 0$ pour $j \leq n$ car il n'y a pas de bénéfice sans magasin.

Questions suivantes hors TD (pour entraînement à domicile en préparation aux prochains cours)

Question 12 Ecrire un algorithme récursif avec mémoïsation qui calcule le bénéfice optimal $B(n, m)$ en temps polyomial en n et m. Quelle est la place mémoire allouée ? Quel est le travail ? Préciser le nombre d'opérations (en $\Theta()$).

Recursif avec mémoïsation : on tabule les résultats de tous les appels $B(i, j)$ pour $0 \leq i \leq n$ et $1 \leq j \leq m$ dans une table de hachage avec comme clef d'accès (i, j).. La place mémoire est $(n + 1)m$.
Travail : le combre d'opérations est de l'ordre du nombre de comparaisons pour le calcul du gain max. Le corps de l'appel $B(i, j)$ (hors appel récursif) fait i comparaisons de gain (max de $i + 1$ éléments). Chaque appel n'est fait qu'une seule fois et son résultat est tabulé. En pire cas, on fait tous les appels donc un travail total en nombre de comparaisons $W(n, m) = \sum_{i=0}^{n} \sum j = 1^m i = \frac{mn(n+1)}{2} = O(n^2 m)$.

Question 13 Analyser le nombre de défauts de cache de ce programme récursif.

Avec la table de hachage, pour une fonction de hachage quelconque, les accès à la table n'ont pas de localité : on fait donc $O(n^2 m)$ défauts de cache.
Mais si, pour $1 \leq j \leq m$ et $0 \leq i \leq n$, on stocke $B(i, j)$ dans une table T à l'emplacement $T[(j-1) * m + i]$, alors le cors de l'appel de $B(i, j)$ fait n accès contigüs dans la table T, donc $\frac{n}{L}$ défauts par appel, soit en tout $\frac{n^2 m}{L}$ défauts.

Question 14 Analyser les dépendances entre les instructions ; en déduire un programme itératif qui calcule la valeur $B(n, m)$ en utilisant un espace mémoire $O(n)$. Préciser le nombre d'opérations (en $\Theta()$).

Itératif sur le nombre de magasins (de 0 à sm) avec une boucle interne sur les paniers à répartir. En faisant attention au stockage pour parcourir des éléments contigüs, le programme itératif s'écrit :

```
for (int j=1 ; j <= m ; ++j)
{  for (int i=0; i <= n ; ++i )
   {  for (int k=0 ; k<i; ++k)  /* Calcul max */
      {
         // .. acces T[j-1,k] et maj T[j,i]...
}  }  }
```

Pour l'implémentation, on utilise 2 tableaux tampons `old` et `new` que l'on alterne : $T[j-1, k]$ est stocké dans `old[k]` et $T[j, i]$ dans `new[i]`. La place mémoire requise est donc $2m$.
Remarque : on peut faire avec un seul tableau mais ce n'est pas une optimisation critique car de toute façon il faudra une mémoire ωnm pour mémoriser les chemins optimaux !

Question 15 Analyse le nombre de défauts de cache de ce programme itératif.

La boucle sur k fait $\frac{i}{L}i$ défauts, donc en tout $O\left(\frac{n^2m}{L}\right)$ défauts.

Question 16 Peut-on a priori diminuer ce nombre de défauts de cache et si oui comment ? (On ne demande pas de programme juste une estimation a priori de ce qui est atteignable).

On utilise une technique de blocking : on regoupe les instructions sur i et j par blocs respectifs de K_1 et K_2 : on a des blocs de $K_1 \times K_2$ instructions qui font $K_1 \times K_2$ accès et tiennent en cache si $K_1 \times K_2 < Z$. Le nombre de défauts de cahe est alors $\sum_{I=0}^{n/K_1} \sum_{J=1}^{n/K_2} i\frac{K_1K_2}{L} \simeq \frac{n^2M}{K_1L}$. Pour minimiser les défauts, on chosit donc $K_1 = Z$ et $K_2 = 1$; le nombre de défauts est $O\left(\frac{n^2m}{LZ}\right)$ défauts. Pour être cache oblivious, la boucle est faite récursivement (similaire au schéma double boucle imbriqué vu en cours 2).