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 $s = 0, \ldots, 10$?

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.

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) ?

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).

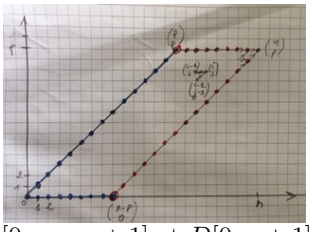
Question 5 Ecrire un programme itératif sans calcul redondant qui calcule et stocke tous les coefficients binomiaux utillisés dans le calcul de $\binom{n}{p}$; préciser le nombre d'opérations (par exemple additions) et la place mémoire requise.

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).

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.

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 $0 \leq k \leq K_1$) et les K_2 coefficients $\binom{I+k}{J+k}$ pour $0 \leq k \leq K_2$).



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

```
1  Integer* L=0 ; // Assumed preallocated and intialized
2  Integer* D=0 ; // Assumed preallocated and intialized
3
4  void calculBloc(int I, int J, int K1, int K2)  // Precondition: 0 <= J <= I
5     /* A l'entrée de ce bloc:
6      *     L[I+k] stocke C(I+k, J) pour 0 <= k <= K1
7      *     D[J+k] stocke C(I+k, J+k) pour 0 <= k <= K2
8      * A la sortie du bloc:
9      *     L[I+k] stocke C(I+k, J+K2-1) pour 0 <= k <= K1
10     *     D[J+k] stocke C(I+K1-1+k, J+K1-1+k) pour 0 <= k <= K2
11     */
12 { for (int j=J+1; j <= J+K2 ; ++j)
13    {  D[j-1] = L[I+K1] ;
14       L[I] = D[j];
15       for (int i=I+1; i <= I+K1; ++i) L[i] = L[i-1]+L[i] ; // acces contigus
16    }
17    D[J+K2] = L[I+K1];
18 }
```

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

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.

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

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

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

Question 11 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 $\Theta()$).

Question 12 Analyser le nombre de défauts de cache de ce programme récursif.

Question 13 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 $\Theta()$).

Question 14 Analyse le nombre de défauts de cache de ce programme itératif.

Question 15 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).