TD 2
Algorithmes de tri

Polytech Marseille - IRM 3ème année
Alexandra Bac

Algorithmique et structures de données
2h TD - 2h TP

Tout le TD sera réalisé en pseudo-code.

1 TD

Exercice 1 (Quicksort)**.** Soit `T` un tableau de `n` entiers.

(i) Mettre en place en pseudo-code l'algorithme de quicksort. En particulier, pourquoi est-on sûr que chaque appel récursif converge ?

(ii) Quel serait le choix optimal de pivot ? Est-ce que son calcul est simple ? Que coûterait son calcul naïf (penser au tri par sélection/échange) ? A-t-on intérêt à l'utiliser ?

Exercice 2 (Tri fusion)**.** Soit `T` un tableau de `n` entiers.

(i) Mettre en place en pseudo-code l'algorithme de tri fusion.

(ii) Quel est le meilleur / pire des cas, le cas moyen pour la complexité ?

(iii) Quel espace mémoire est nécessaire à cet algorithme ? Comparer avec le quicksort.

Exercice 3 (Tri bulle)**.** Soit `T` un tableau de `n` entiers.

(i) A partir des principes de tri bulle vus en cours, écrire le pseudo-code le plus efficace possible pour le tri bulle.

(ii) Comment améliorer encore ? (tri shaker) Est-ce que cela change la complexité moyenne ?

2 TP

Implémenter en C ces algorithmes de tri et comparer leurs performances.

3 Challenge

La programmation dynamique est une approche permettant de calculer de manière efficace des solutions optimales à des problèmes combinatoires / discrets. On cherche la meilleure solution à un problème, et le "meilleur" est mesuré par une fonction de valeur ou de coût. La programmation dynamique consiste à exprimer récursivement la meilleure solution à partir d'une sous-solution. Le point étrange est qu'elle calcule la *valeur* de la solution optimale avant d'en déduire la solution elle-même ...

Pour mieux décrire l'approche, j'ai choisi de traiter un problème comme exemple (problème du sac à dos), puis de vous laisser en résoudre un autre (problème du parenthésage du produit de matrices).

3.1 Problème du sac à dos par programmation dynamique

On s'intéresse au problème suivant : on considère un sac à dos de volume V entier et n objets ayant chacun un volume v_i entier et une valeur x_i. On veut déterminer le choix optimal d'objet (ayant la valeur totale la plus grande) rentrant dans le sac à dos (donc tels que la somme des volume reste inférieure à V). C'est un problème compliqué ...

Une solution au remplissage du sac à dos est le choix d'un sous-ensemble d'objets $I \subseteq \{1 \dots n\}$ tel que

$$\sum_{i \in I} v_i \leqslant V$$

La solution optimale est celle pour laquelle, de plus, la valeur de la solution $\sum_{i \in I} x_i$ est maximale.

On notera $f(n, V)$ la valeur de la solution optimale pour les objets $1 \dots n$ et le volume V.

L'approche de la programmation dynamique comporte les étapes suivantes :

(i) Exprimer la valeur de la solution optimale, $f(n, V)$, récursivement

(ii) On va en déduire le remplissage d'une matrice A de taille $(n+1) \times (V+1)$ contenant dans la case (m, W) (avec $m \leqslant n$ et $W \leqslant V$) la valeur de $f(m, W)$.

(iii) On crée en parallèle une seconde matrice B de même taille dans laquelle on mémorise dans chaque case (m, W) le choix ayant conduit à cette valeur optimale en fonction des valeurs précédentes

(iv) La solution optimale est alors reconstruire à partir de B par "décodage" de l'information qu'elle contient

Etape 1. Une solution optimale au problème (n, V) (ie. pour n objets et volume V) est en fait :
- Un choix pour l'objet n (pris ou pas pris)
- Puis :
  - Si l'objet n n'est pas pris, alors la solution est en fait la solution optimale pour $(n-1, V)$
  - Si l'objet n est pris, alors la solution est composée : d'une solution optimale pour $(n-1, V-v_n)$ plus l'objet n

On obtient donc la formule récursive suivante pour les valeurs de solutions optimales :

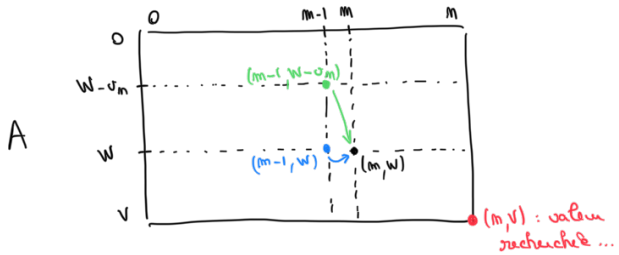
$$f(n, V) = \max\left(f(n-1, V),\ x_n + f(n-1, V-v_n)\right) \tag{1}$$

Avec les conditions particulières suivantes :
- $f(n, V) = 0$ si $n = 0$ ou $V = 0$
- $f(n, V) = f(n-1, V)$ si $v_n > V$

Etape 2. Pour calculer $f(n, V)$, on va en fait remplir toute la matrice A des $f(m, W)$ avec $0 \leqslant m \leqslant n$ $0 \leqslant W \leqslant V$. Cela peut paraître étrange mais vous avez déjà rencontré cela. Pour calculer les coefficients binomiaux (les C_n^k), on a la formule récursive : $C_{n+1}^{k+1} = C_n^k + C_n^{k+1}$. Mais on sait très bien qu'il ne faut surtout pas lancer le calcul récursif car les coefficient intermédiaires seraient calculés plusieurs fois ... On calcule donc les coefficients binomiaux en remplissant le tableau des C_m^j pour $j \leqslant k$ est $m \leqslant n$: le triangle de Pascal ... Ainsi chaque coefficient est calculé une seule fois.

Tout le problème est de déterminer les dépendances récursives par rapport à 1 entre les cases de la matrice. On pourra ainsi décider d'un ordre de calcul des cases respectant l'ordre récursif.

Dans notre problème, les cas terminaux correspondent à $n = 0$ ou $V = 0$ (donc la première ligne et la première colonne de la matrice). Et la dépendance récursive peut être représentée de la manière suivante :



Si fait les calculs ligne par ligne ou colonne par colonne, les dépendances récursives sont donc satisfaites : une case est bien calculée après les cases dont elle dépend.

On est maintenant en mesure de remplir toute la matrice A simplement par deux boucles "for" imbriquées.

Etape 3. Toute la question est maintenant d'arriver à déterminer la solution optimale (et pas seulement sa valeur, calculée dans A) ... Pour cela, on remarque que chaque case est le maximum de deux cases précédemment calculées. Et savoir si c'est la première (verte) ou la deuxième (bleue) qui a donné le maximum correspond à savoir si l'objet m a été pris ou refusé.

La matrice B stocke simplement cette information. On utilisera ici le codage suivant : sa case (m, W) contient 1 si c'est la case verte qui donne le maximum (ie. l'objet m est pris) et 0 sinon (ie. l'objet m est refusé).

L'algorithme de remplissage conjoint des matrices A et B est donc :

```
// Initialisation de A
pour W de 0 à V
    A(0,W) ← 0
fpour
pour m de 0 à n
    A(m,0) ← 0
fpour
```

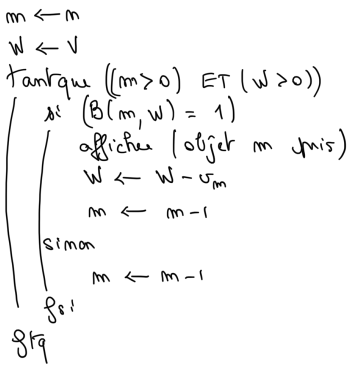
```
// Remplissage par ligne
pour W de 1 à V
    pour m de 1 à n
        si (v_m > W)
            A(m,W) ← A(m-1,W)
        sinon
            pris ← x_m + A(m-1,W-v_m)
            refusé ← A(m-1,W)
            si (refusé < pris)
                A(m,W) ← pris
                B(m,W) ← 1
            sinon
                A(m,W) ← refusé
                B(m,W) ← 0
            fsi
        fsi
    fpour
fpour
```

Sa complexité est $\mathcal{O}(n, V)$.

Etape 4. Ne reste plus qu'à décoder ... Pour construire la solution (n, V), on commence par consulter la case $B(n, V)$:

- Si elle contient un 1, l'objet n est pris et la suite de la solution optimale est celle du problème $(n-1, V-v_n)$. On va donc à la case $B(n-1, V-v_n)$.
- Sinon, l'objet n n'est pas pris et la suite de la solution optimale est celle du problème $(n-1, V)$. On va donc à la case $B(n-1, V)$.
- etc. ...

Le pseudo-code correspondant est alors :



3.2 A vous ...

Exercice challenge (Programmation dynamique et parenthésage de produits de matrices)**.** Dans cet exercice, on s'intéresse au parenthésage optimal d'une suite de produits de matrices.

(i) Tout d'abord, remise des idées en place. Soit A une matrice $n \times m$ et B une matrice $m \times p$, la matrice produit $A \times B$ est de taille $n \times p$. Justifier pourquoi le produit coûte nmp multiplications.

(ii) Ensuite, petite prise de conscience : on considère trois matrices A, B, C, de tailles respectives 2×4, 4×8 et 8×3. Combien de multiplications sont nécessaires pour calculer $(A \times B) \times C$ et $A \times (B \times C)$ respectivement ?

(iii) Le but de cet exercice est de déterminer le parenthésage optimal d'un produit de n matrices $A_1 \times \cdots \times A_n$. Chaque matrice A_i est de taille $n_i \times n_{i+1}$.
Soient $i \leqslant j$, on note $p(i,j)$ le nombre de la multiplications pour le parenthésage optimal de $A_i \times \cdots \times A_j$. On cherche donc $p(1,n)$.

(a) **Etape 1** : expression récursive de $p(i,j)$ avec $i \leqslant j$. Comment est constitué un parenthésage de $A_i \times \cdots \times A_j$? En déduire, lorsque $j - i \geqslant 2$, une expression de la forme :

$$p(i,j) = \min_{k=i\ldots j-1} (p(\cdots) + p(\cdots) + n_i n_{k+1} n_{j+1}) \tag{2}$$

(b) Il faut maintenant exprimer les conditions d'arrêt. Que se passe-t-il quand :
— $j = i + 1$?
— $j = i$?

(c) **Etape 2** : la matrice A contient $p(i,j)$ dans sa (i,j)ème case. On a $1 \leqslant i \leqslant j \leqslant n$. Si A est de taille $n \times n$ (avec indices commençant à 1), représenter la portion de la matrice utilisée pour stocker ces coefficients. Représentez également les coefficients déterminés par les conditions d'arrêt.

(d) Dans quel ordre faut-il calculer les lignes de A pour que l'ordre des calculs respecte les dépendances récursives ?

(e) **Etape 3** : quelle information faut-il stocker dans B pour retrouver l'information correspondant au calcul du minimum de l'équation 2 ?

(f) En déduire un algorithme de calcul de A et B (il comprend forcément une initialisation ...)

(g) **Etape 4** : puis en déduire un algorithme récursif de calcul du parenthésage optimal par décodage de B.

(h) En utilisant le matériel fourni (implémentant des matrices bi-dimensionnelles), implémenter et afficher le parentésage optimal. L'affichage se fera sous la forme : `((1,2),(3,4))` par exemple pour un produit de 4 matrices, donc en indiquant simplement les indices des matrices (parmi $1 \ldots, n$). Vous afficherez également le nombre de produits nécessaires.

Lisez bien la section suivante pour savoir comment utiliser ce matériel et compiler. La programmation modulaire sera vue plus précisément au chapitre suivant.

Votre code sera découpé en fonctions propres, pas trop grosses, qui seront appelées depuis le `main`. Par exemple, une fonction d'initialisation de la matrice A, une fonction de remplissage (avec la formule récursive) de A et B, et une fonction de "décodage" et reconstruction du parenthésage optimal.

(i) Tester le parenthésage optimal d'un produit de 3 matrices des tailles :

$$3 \times 2, 2 \times 4, 4 \times 5$$

Vous devriez obtenir `(1, (2, 3))` et un nombre de produits de 70.

(j) Déterminer le parenthésage optimal du produit de 10 matrices de tailles :

$$3 \times 8, 8 \times 4, 4 \times 5, 5 \times 2, 2 \times 10, 10 \times 20, 20 \times 4, 4 \times 3, 3 \times 6, 6 \times 12$$

Vous utiliserez des matrices de taille $(n + 1) \times (n + 1)$ dont vous n'utiliserez pas la première ligne/colonne (d'indice 0).

3.3 Utilisation du matériel d'exercice challenge

Le matériel fourni utilise la programmation modulaire ; cela consiste à découper le code en parties distinctes (chacune traitant un morceau du problème : matrices bi-dimensionnelles, code dynamique). L'intérêt est double : 1) pouvoir organiser le code en morceaux cohérents, 2) pouvoir compiler chacun séparément (donc ne pas tout recompiler à chaque fois).

Chaque module est composé d'une partie d'entêtes (fichier.h) et d'une partie code (fichier.c). Donc les matrices bi-dimensionnelles fournies comprennent ces deux fichiers : `mymatrix.h` et `mymatrix.c`.

Comme la compilation devient alors plus complexe (compiler chaque module séparément et regrouper le tout en un exécutable), elle est automatisée par un fichier décrivant ce qu'il faut compiler, comment et les dépendances : le `Makefile`. Ce fichier est situé (dans un premier temps) dans le répertoire où se trouvent les fichiers de code (.c et .h). Il suffit, dans un terminal d'exécuter `make` pour lancer la compilation.

Vous mettrez votre code (fonctions et main) dans le fichier `parenthesages.c`. Pour coder les matrices A et B, le module `mymatrix` vous fournit un type de matrices bi-dimensionnel `MyMatrix` ainsi que les fonctions suivantes :

```
// Création d'une matrice de taille nl x nc
MyMatrix create_matrix(int nl, int nc) ;
// Desallocation d'une matrice bi-dimensionnelle A
void free_matrix(MyMatrix A) ;

// Affectation d'une valeur à un coefficient, ie stockage de e dans A_ij
void set_matrix(MyMatrix A, int i, int j, Data e) ;
// Récupération de la valeur A_ij
Data get_matrix(MyMatrix, int, int) ;

// Affichage de la matrice
void print_matrix(MyMatrix) ;
```

Les coefficients sont tous initialisés à 0 par défaut.

Donc par exemple pour créer une matrice identité et récupérer les coefficients d'une ligne, le main contiendra :

```
MyMatrix A = create_matrix(4,4) ;
int i, j ;
```

```
for(i=0; i<4; ++i)
{
    set_matrix(A,i,i,1) ;
}
// Affichage de la ligne 0
for(j=0; j<4; ++j)
    printf("%d ", get_matrix(A,0,j)) ;
printf("\n");
return 0 ;
```