IN 101 - TD 07
21 octobre 2011

Corrigé

ENSTA ParisTech

Solution 1 : Gestion des matrices

1.a] On peut utiliser le fait qu'une affectation retourne la valeur affectée pour tester directement que le `malloc` a marché (ligne 9 du code).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  int n;
5
6  int** matinit() {
7    int i,j;
8    int** res = (int**) malloc(n*sizeof(int*));
9    if (!res) {
10     printf("Erreur d'allocation!\n");
11     return NULL;
12   }
13   for (i=0; i<n; i++) {
14     if (!(res[i] = (int*) malloc(n*sizeof(int)))) {
15       printf("Erreur d'allocation!\n");
16       return NULL;
17     }
18     for (j=0; j<n; j++) {
19       res[i][j] = rand() % 10;
20     }
21   }
22   return res;
23 }
```

1.b] Voici le code d'affichage, une simple double boucle.

```
1  void matprint(int** mat) {
2    int i,j;
3    for (i=0; i<n; i++) {
4      for (j=0; j<n; j++) {
5        printf("%4d ", mat[i][j]);
6      }
7      printf("\n");
8    }
9    printf("\n");
10 }
```

1.c] La fonction de libération (à chaque `malloc` dans `matinit` correspond un `free` ici) et un exemple de `main` qui crée une matrice, l'affiche et la libère.

```
1  void matfree(int** mat) {
2    int i;
3    for (i=0; i<n; i++) {
4      free(mat[i]);
5    }
6    free(mat);
7  }
8
9  int main(int argc, char* argv[]) {
10   int** m;
11   if (argc != 2) {
12     printf("Donner la taille des matrices en argument.\n");
13     return 1;
14   }
15   srand(getpid());  // on initialise la graine une seule fois
16   n = atoi(argv[1]);
17
18   m = matinit();
19   matprint(m);
20   matfree(m);
21   return 0;
22 }
```

Solution 2 : Quelques opérations

Ces trois opérations sont très simples à programmer, il suffit de quelques boucles `for` imbriquées.

```
1  int** matadd(int** a, int** b) {
2    int i,j;
3    int** res = matinit();
4    for (i=0; i<n; i++) {
5      for (j=0; j<n; j++) {
6        res[i][j] = a[i][j] + b[i][j];
7      }
8    }
9    return res;
10 }
11 int** matmul(int** a, int** b) {
12   int i,j,k;
13   int** res = matinit();
14   for (i=0; i<n; i++) {
15     for (j=0; j<n; j++) {
16       res[i][j] = 0;           // la case (i,j) est la produit scalaire de
17       for (k=0; k<n; k++) {    // la ligne i de a et la colonne j de b
18         res[i][j] += a[i][k] * b[k][j];
19       }
20     }
21   }
22   return res;
23 }
```

```
24 int** mattrans(int** a) {
25   int i,j;
26   int** res = matinit();
27   for (i=0; i<n; i++) {
28     for (j=0; j<n; j++) {
29       res[i][j] = a[j][i];
30     }
31   }
32   return res;
33 }
34
```

Solution 3 : Calcul du déterminant

3.a] Voici la fonction récursive pour la calcul du déterminant : la fonction fait n appels récursifs et pour chaque appel elle crée un nouveau tableau de n-1 lignes. Ce tableau de n-1 lignes n'est alloué qu'une fois au début et libéré une fois tous les sous-détereminants calculés.

```
1  int matdet(int** mat, int n) {
2    int i,j,det;
3    int** submat;
4    if (n == 1) { // condition d'arrêt
5      return mat[0][0];
6    }
7    submat = (int**) malloc((n-1)*sizeof(int));
8    det = 0;
9    for (i=0; i<n; i++) {
10     // on retire la i-ème ligne en recopiant en 2 fois
11     for (j=0; j<i; j++) {
12       submat[j] = mat[j];  // on recopie le pointeur vers la ligne
13     }
14     for (j=i+1; j<n; j++) {
15       submat[j-1] = mat[j];
16     }
17     if ((n-1+i)%2 == 0) {  // on test la parité de la puissance du -1
18       det += mat[i][n-1] * matdet(submat, n-1);
19     } else {
20       det -= mat[i][n-1] * matdet(submat, n-1);
21     }
22   }
23   free(submat);    // bien penser à libérer la mémoire allouée
24   return det;
25 }
```

3.b] Ici le calcul du déterminant coûte $\Theta(n!)$. On peut le faire en $\Theta(n^3)$ avec un pivot de Gauss.

3.c] Voici le code du pivot de Gauss. Il faut bien faire attention à ne faire que des calculs d'entiers, donc on garde en permanence un coefficient multiplicatif par lequel on divise le déterminant à la fin. Pour éviter que les entiers dans la matrice ne grossissent trop, on doit réduire les lignes en calculant leur PGCD à chaque fois (mais les coefficients grossissent quand même vite).

```
int gcd(int a, int b) {
  int r;
  while (b !=0) {
    r = a%b;
    a = b;
    b = r;
  }
  return a;
}

int gauss(int** mat) {
  int i,j,k, coeff, res, *tmp, pgcd;
  int** gmat = matinit();
  // on commence par recopier la matrice pour ne pas la modifier
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      gmat[i][j] = mat[i][j];
    }
  }
  // on commence le pivot, on divisera le déterminant par coeff à la fin
  coeff = 1;
  for (i=0; i<n; i++) {   // on veut annuler la colonne i
    // on cherche une ligne avec un coefficient non nul
    for (j=i; j<n; j++) {
      if (gmat[j][i] != 0) {
        break;           // on quitte la boucle
      }
    }
    // si la boucle arrive à la fin sans coef non nul, le déterminat est 0
    if (j == n) {
      return 0;
    }
    if (j != i) {         // il faut échanger 2 lignes
      tmp = gmat[i];
      gmat[i] = gmat[j];
      gmat[j] = tmp;
      if ((j+i)%2 != 0) { // si la distance est impaire
        coeff = -coeff;
      }
    }
    // maintenant on va soustraire la i-ème ligne de la matrice aux autres lignes
    for (j=i+1; j<n; j++) {
      coeff *= gmat[i][i];   // la j-ème ligne sera multipliée par gmat[i][i]
      for (k=i+1; k<n; k++) {
        gmat[j][k] = gmat[j][k]*gmat[i][i] - gmat[j][i]*gmat[i][k];
      }
      gmat[j][i] = 0;
      // on divise la j-ème ligne (et coeff) par le pgcd de ses coefficients
      pgcd = coeff;
      for (k=i+1; k<n; k++) {
        pgcd = gcd(pgcd, gmat[j][k]);
      }
      coeff /= pgcd;
      for (k=i+1; k<n; k++) {
        gmat[j][k] /= pgcd;
      }
    }
  }
  // on calcule le produit des coefficients diagonaux
  res = 1;
```

```
61   for (i=0; i<n; i++) {
62     res *= gmat[i][i];
63   }
64   // on divise par le coefficient accumulé pendant le pivot
65   res = res/coeff;
66   matfree(gmat);  // on libère la matrice temporaire
67   return res;
68 }
```

Solution 4 : Multiplication de Strassen (pour ceux qui vont très vite)

4.a] Avec l'algorithme naïf, le nombre d'additions est $n^2(n-1)$ et le nombre de multiplications n^3.

4.b] Dans l'algorithme naïf, il y a 8 multiplications et 4 additions, alors que dans l'algorithme de Strassen il y a 7 multiplications et 18 additions.

4.c] On a m^3 multiplications pour chaque p_i (chaque coefficient est maintenant un bloc de $m \times m$), d'où $Mult(n) = 7m^3 = \frac{7}{8}n^3$ et pour les additions, il y a les p_i, donc $7m^2(m-1)$ et les 18 matrices auxiliaires qui contribuent pour $18m^2$, soit $Add(n) = 7m^3 + 11m^2 = \frac{7}{8}n^3 + \frac{11}{4}n^2$.

4.d] On a $M(1) = 1$ et $M(n) = 7 \times M(n/2)$ donc $M(n) = 7^{\log_2(n)} = n^{\log_2(7)}$. On a $A(1) = 0$ et $A(n) = 7 \times A(n/2) + 18 \times (n/2)^2$ donc $A(n) = 6 \times (n^{\log_2(7)} - n^2)$. Le plus important est que $\log_2(7) \simeq 2.807$ est plus petit que 3, donc on passe d'un produit qui coûte $\Theta(n^3)$ à un produit en $\Theta(n^{2.8})$. En revanche, pour n petit, le coût des additions à effectuer n'est pas négligeable et la méthode naïve peut être plus rapide.

4.e] Désolé, pas de corrigé pour l'instant...