DM2 MP2I : allocations

Thèmes

1. Paradigme « Diviser pour Régner » ;
2. Allocation dynamique ;
3. Fonctions récursives.

Présentation

Ce devoir est l'occasion de se familiariser avec l'allocation dynamique. Nous allons manipuler des pointeurs en réservant puis en libérant de l'espace sur le tas.

Afin de contrôler que les codes proposés font autant d'allocations que de libération, nous introduisons deux variables globales à déclarer au début du fichier rendu :

```c
int nballoc = 0;
int nbfree = 0;
```

Les allocations et libérations se font exclusivement avec les deux fonctions :

```c
// avec incrémentation de nballoc
void *mymalloc(size_t size){
  nballoc++;
  return malloc(size);
}
//avec incrémentation de nbfree
void myfree(void *ptr)void myfree(void *ptr){
  nbfree++;
  free(ptr);
}
```

Aucun appel aux fonctions `malloc`, `calloc`, `free` et `realloc` n'est donc plus autorisé.

Comme nous en prenons l'habitude, la fonction `main` ne comporte que quelques appels à des fonctions tests. Elle se termine cependant par une comparaison des variables globales `nballoc` et `nbfree` et le déclenchement d'une erreur d'assertion si les deux quantités ne sont pas égales :

```c
int main(){
  test_strassen();
  printf("nballoc =%d, nbfree=%d\n",nballoc , nbfree);
  assert(nballoc == nbfree);
  return 0;
}
```

Un code correct ne doit pas déclencher d'erreur en avant dernière ligne.

Multiplication de Strassen

Pour faire ce devoir, il convient de se documenter sur le *produit matriciel* tel qu'il est abordé en cours de mathématiques (voir les sections « Produit matriciel ordinaire » et « Exemples » ici). Ce produit s'implémente bien par une triple boucle imbriquée.

La *multiplication de Strassen* est un procédé qui diminue la quantité de multiplications de nombres dans un produit matriciel par rapport à l'algorithme dit « naïf » (celui du cours de mathématiques). Cet algorithme respecte le paradigme *diviser pour régner* (voir ici). On trouve une bonne explication de cette méthode à la section « Description de l'algorithme » sur cette page.

L'étude de la complexité en mémoire et en temps de cet algorithme fera l'objet d'un point de cours ultérieurement. Dans ce devoir, les vecteurs sont représentés par des `int *` obtenus dynamiquement et les matrices par des `int **` obtenus de la même façon.

Travail à faire

Pour toute fonction demandée, on présente dans ce qui suit une fonction de tests appelée dans le `main` et le rendu obtenu dans un terminal après compilation et exécution. On observe bien que le nombre d'allocations est égal à celui de libérations.

Q1 Écrire les 5 fonctions :

(a) `int* make_vect(int n, int x)` qui retourne un tableau dynamique de 4 entiers (un *vecteur*) dont les n coefficients valent x.

(b) `void display_vect(int n, int* v)` qui affiche le contenu d'un vecteur.

(c) `int** make_mat(int n, int m, int x)` qui retourne un tableau dynamique représentant une matrice de n lignes de m colonnes dont tous les coefficients valent x.

(d) `void display_mat(int n, int m, int** mat)` qui affiche le contenu d'une matrice en alignant bien les colonnes.

(e) `void freemat(int n, int **M)` qui libère tous les pointeurs qui permettent de représenter la matrice M et `M` lui-même.

```c
void test_display_make_vect(){
  int *p = make_vect(5,1);
  display_vect(5,p);
  myfree(p);
}

void test_display_make_mat(){
  int **p = make_mat(5,3,2);
  display_mat(5,3,p);
  freemat(5,p);
}

int main(){
  test_display_make_vect();
  test_display_make_mat();
  printf("nballoc =%d, nbfree=%d\n",
    nballoc , nbfree);
  assert(nballoc == nbfree);
  return 0;
}
```

Listing 1 – Tests

Listing 2 – Rendu sur console

```
$ ./a.out
    1    1    1    1    1
    2    2    2
    2    2    2
    2    2    2
    2    2    2
    2    2    2
nballoc =7, nbfree=7
```

Q2 Écrire la fonction `int** cl(int n, int m, int alpha, int** M1, int** M2)` qui calcule la combinaison linéaire $M_1+\alpha M_2$ pour les deux matrices de mêmes dimensions M_1, M_2 et pour le nombre α.

En prenant $\alpha = 1$ ou $\alpha = -1$, on est donc en mesure d'évaluer l'addition ou la soustraction de deux matrices.

```c
void test_cl(){
  int n=3;
  int **mat1 = make_mat(n,n,0),
  **mat2 = make_mat(n,n,0);
  for (int i=0;i<n;i++){
    mat1[i][i]=i+1;
    mat2[i][n-1-i]=3-i;
  }
  printf("mat1 = \n");
  display_mat(n,n,mat1);
  printf("mat2 = \n");
  display_mat(n,n,mat2);
  printf("mat1 - mat2 = \n");
  int** p= cl(n,n,-1,mat1,mat2);
  display_mat(n,n,p);
  freemat(n,mat1);freemat(n,mat2);
    freemat(n,p);
}

```

Listing 3 – Fonction de tests à placer dans `main`

Listing 4 – Rendu sur console

```
$ ./a.out
mat1 =
    1    0    0
    0    2    0
    0    0    3
mat2 =
    0    0    3
    0    2    0
    1    0    0
cl(n,n,-1,mat1,mat2) =
    1    0   -3
    0    0    0
   -1    0    3
nballoc =12, nbfree=12
```

Q3 Écrire la fonction `int ** naif_mult(int n, int m, int p, int **M1, int **M2)` qui réalise le produit de la matrice M_1 de dimensions $n \times m$ par la matrice M_2 de dimensions $m \times p$. L'algorithme est celui du cours de mathématiques. Il est expliqué ici.

```c
void test_naif_mult(){
  int n=2,m=3,p=4;
  int** mat1 = make_mat(n,m,0);
  int** mat2 = make_mat(m,p,0);

  mat1[0][0]=1;mat1[0][2]=-1;
  mat1[1][0]=2;mat1[1][1]=1;

  mat2[0][0]=1;mat2[0][2]=-1;
  mat2[1][0]=2;mat2[1][3]=1;
  mat2[2][1]=-1;mat2[2][2]=1;mat2[2][3]=1;

  printf("mat1 = \n");
  display_mat(n,m,mat1);
  printf("mat2 = \n");
  display_mat(m,p,mat2);
  printf("mat1 * mat2 = \n");
  int ** res = naif_mult(n,m,p,mat1,mat2);
  display_mat(n,p,res);
  freemat(n,mat1);freemat(m,mat2);freemat(n,res);
}
```

Listing 5 – Fonction de tests à placer dans `main`

Listing 6 – Rendu sur console

```
$ ./a.out
mat1 =
   1   0  -1
   2   1   0
mat2 =
   1   0  -1   0
   2   0   0   1
   0  -1   1   1
mat1 * mat2 =
   1   1  -2  -1
   4   0  -2   1
nballoc =10, nbfree=10
```

Dans la suite du devoir, nous manipulons des matrices carrées de tailles $n \times n$ où n est une puissance de 2. Il faut d'abord traiter le problème des matrices qui ne sont pas de cette nature.

Q4 Si le nombre de lignes n'est pas une puissance de 2, on complète la matrice par des coefficients nuls.

(a) Écrire la fonction `int log_2(int n)` qui calcule le plus petit e tel que $2^e \geq n$. On peut avec profit utiliser les opérateurs bitwise.

(b) Écrire la procédure `void resize(int n,int*** mat)` qui ajoute des 0 à une matrice de taille $n \times n$ de façon à ce que ses dimensions soient $2^e \times 2^e$ où e est obtenu par la fonction précédente. Cette question, délicate, peut être négligée sans préjudice pour la suite.

```c
void test_log_2(){
  int tab[3] = {15,16,17};
  for (int i =0; i<3;i++)
    printf("log_2(%d)=%d\n",tab[i], log_2(tab[i]));
}

void test_resize(){
  int n = 3;
  int m = 1<<log_2(n);
  int** mat1 = make_mat(n,n,0);
  for(int i = 0; i<n;i++)
      for(int j = 0; j<n;j++)
    mat1[i][j]=i*n +j;
  printf("mat1 = \n");  display_mat(n,n,mat1);
  printf("resize(mat1)\n"); resize(n,&mat1);
  printf("mat1 = \n");  display_mat(m,m,mat1);
  freemat(m,mat1);
}
```

Listing 7 – Fonctions de tests à placer dans `main`

Listing 8 – Rendu sur console

```
$ ./a.out
log_2(15)=4
log_2(16)=4
log_2(17)=5
mat1 =
    0    1    2
    3    4    5
    6    7    8
resize(mat1)
m=4
mat1 =
    0    1    2    0
    3    4    5    0
    6    7    8    0
    0    0    0    0
nballoc =9, nbfree=9
```

Dorénavant, toutes les matrices sont carrées avec un nombre de lignes égal à une puissance de 2.

Q5 Écrire la fonction `int *** fourblocks(int n, int **mat)`. Cette fonction décompose une matrice de taille $2^e \times 2^e$ en 4 blocs de tailles $2^{e-1} \times 2^{e-1}$. Ces 4 blocs sont ensuite placé dans le tableau de matrices (`int ***`) qui est retourné.

```c
void test_fourblocks(){
  int n = 4;
  int** mat = make_mat(n,n,0);
  for(int i = 0; i<n;i++)
      for(int j = 0; j<n;j++)
    mat[i][j]=i*n +j;
  printf("mat = \n");
  display_mat(n,n,mat);
  int *** t = fourblocks(n,mat);
  for(int i = 0; i<4;i++){
    printf("t[%d] = \n",i);
    display_mat(n/2,n/2,t[i]);
  }
  for(int i = 0; i<4;i++)
    freemat(n/2,t[i]);
  freemat(n,mat);
  myfree(t);
}
```

Listing 9 – Fonction de tests à placer dans `main`

Listing 10 – Rendu sur console

```
$ ./a.out
mat =
    0    1    2    3
    4    5    6    7
    8    9   10   11
   12   13   14   15
t[0] =
    0    1
    4    5
t[1] =
    2    3
    6    7
t[2] =
    8    9
   12   13
t[3] =
   10   11
   14   15
nballoc =18, nbfree=18
```

Q6 Écrire la fonction `void merge(int n, int ** t[4],int **m)` qui réalise l'opération inverse de la précédente. Elle prend en paramètre un tableau contenant 4 matrices de tailles $n \times n$ et « fusionne » ces matrices en une seule de taille $2n \times 2n$.

```c

void test_merge(){
  int n = 4;
  int ** mat = make_mat(n,n,0);
  int ***t =
      mymalloc(4 * sizeof(int**));
  int p = n/2;
  for(int i = 0;i<4;i++){
    t[i]=make_mat(p,p,i);
  }// for i
  // affichage des matrices dans t
  _display_tab_blocs(p,4,t);//Fct à écrire
  // fusion + affichage
  merge(n/2,t,mat);
  printf("mat= \n");
  display_mat(n,n,mat);
  //libérations
  for(int i = 0; i<4;i++)
    freemat(p,t[i]);
  freemat(n,mat);
  myfree(t);

  mat = make_mat(2,2,0);
  t = mymalloc(4 * sizeof(int**));
  t[0]=make_mat(1,1,1);
  t[1]=make_mat(1,1,2);
  t[2]=make_mat(1,1,6);
  t[3]=make_mat(1,1,1);
  _display_tab_blocs(1,4,t);
  merge(1,t,mat);
  printf("mat= \n");
  display_mat(2,2,mat);

  //libérations
  for(int i = 0; i<4;i++)
    freemat(1,t[i]);
  freemat(2,mat);
  myfree(t);
}

```

Listing 11 – Fonction de tests à placer dans `main`

Listing 12 – Rendu sur console

```
Fonction _display_tab_blocs
------- bloc t[0] --------
   0   0
   0   0
------- bloc t[1] --------
   1   1
   1   1
------- bloc t[2] --------
   2   2
   2   2
------- bloc t[3] --------
   3   3
   3   3
mat=
   0   0   1   1
   0   0   1   1
   2   2   3   3
   2   2   3   3
Fonction _display_tab_blocs
------- bloc t[0] --------
   1
------- bloc t[1] --------
   2
------- bloc t[2] --------
   6
------- bloc t[3] --------
   1
mat=
   1   2
   6   1
nballoc =30, nbfree=30
```

Q7 Écrire la fonction **récursive** `int ** strassen(int n, int ** mat1,int ** mat2)` qui réalise le produit de Strassen de deux matrices carrées de dimensions $n \times n$. Cette question, difficile, nécessite l'écriture de plusieurs fonctions auxiliaires. Un soin particulier doit être apporté au nombre d'allocations/libérations : on rappelle que les deux quantités doivent être les mêmes.

Le cas d'arrêt de la récursion intervient uniquement lorsque les matrices passées en paramètres sont de dimensions 1×1 (matrices-coefficients).

```c
void test_strassen(){
  int n = 4;
  int** mat1 = make_mat(n,n,0);
  for(int i = 0; i<n;i++)
      for(int j = 0; j<n;j++)
    mat1[i][j]=i*n +j*j;
  printf("mat1 = \n");
  display_mat(n,n,mat1);

  int** mat2 = make_mat(n,n,0);
  for(int i = 0; i<n;i++)
      for(int j = 0; j<n;j++)
    mat2[i][j]=2*i*i-j;
  printf("mat2 = \n");
  display_mat(n,n,mat2);

  int ** s = strassen(n,mat1,mat2);
  printf("mat1 * mat2 = \n");
  display_mat(n,n,s);

  freemat(n,mat1);   freemat(n,mat2);
  freemat(n,s);
}
```

Listing 13 – Fonction de tests à placer dans `main`

Listing 14 – Rendu sur console

```
$ ./a.out
mat1 =
    0    1    4    9
    4    5    8   13
    8    9   12   17
   12   13   16   21
mat2 =
    0   -1   -2   -3
    2    1    0   -1
    8    7    6    5
   18   17   16   15
mat1 * mat2 =
  196  182  168  154
  308  278  248  218
  420  374  328  282
  532  470  408  346
nballoc =600, nbfree=600
```

Remarque. La multiplication de Strassen n'est intéressante que pour les grosses matrices du fait de la constante multiplicative intervenant dans l'expression de sa complexité. Elle n'est pas non plus adaptée pour les matrices *creuses* (c.a.d avec de nombreux coefficients nuls). De plus, construire explicitement les blocs comme nous le faisons ici est rarement une bonne idée. Il vaut mieux opérer une gestion fine des indices délimitant les différents blocs sans évaluer ces blocs.