Structures de données
TD - 1

1. Exo. 1 - Bibliothèque pour manipuler des matrices (Tableaux de `float` à deux dimensions).
   L'implémentation d'une matrice peut se faire avec deux approches :
   — en utilisant un tableau de tableaux ;
   — en utilisant un unique tableau de $N \times M$ éléments.

   (a) Déterminez comment peut-on implémenter un tableau à deux dimensions (avec N lignes et M colonnes) à partir d'un unique tableau de $N \times M$ éléments ? Quel est l'avantage de cette implémentation par rapport a une implémentation avec des tableau de tableaux ?

   Solution: Il est possible de facilement calculer, a partir des indices $(i, j) \longrightarrow (\text{ligne}, \text{colonne})$, la position de la valeur dans un tableau unique :

   $$p = i * M + j$$

   Les principaux avantages de cette méthode sont qu'elle ne nécessite pas de boucle lors de l'allocation du tableau à deux dimensions et qu'elle ne nécessite pas d'utilisation de double pointeur. De plus, si l'on réfléchie à l'implémentation de tableau à plus de dimensions (3, 4, 5, ... dimensions), l'approche par tableau de tableau de tableau de ... est in-envisageable.

   (b) Pour les deux implémentations possibles, écrivez la structure de donnée pour stocker une matrice de N lignes et M colonnes.

   Solution: Avec un tabeau de tableaux :

```c
struct struct_matrix {
    double ** data;
    size_t N; // Nb lignes
    size_t M;// Nb Colonnes
};
typedef struct struct_matrix s_matrix;
```

   Avec un tableau de $N \times M$ éléments :

```c
struct struct_matrix {
    double * data;
    size_t N; // Nb lignes
    size_t M;// Nb Colonnes
};
typedef struct struct_matrix s_matrix;
```

   (c) Pour les deux implémentations possibles, écrivez la fonction pour allouer une matrice de N lignes et M colonnes.

   Solution: Avec un tabeau de tableaux :

```c
s_matrix * matrix_alloc(size_t N, size_t M){
    s_matrix * p_matrix = (s_matrix *) malloc(sizeof(s_matrix));
    p_matrix->N = N;
    p_matrix->M = M;
```

```
    p_matrix->data = (double **) malloc(sizeof(double *)*N);
    p_matrix->data[0] = (double *) malloc(sizeof(double)*M*N);
    memset(p_matrix->data[0], 0, sizeof(double)*M*N);
    for(size_t i = 1 ; i < N ; i++)
        p_matrix->data[i] = &p_matrix->data[0][M*i];
    return p_matrix;
}
```

Avec un tableau de $N \times M$ éléments :

```
s_matrix * matrix_alloc(size_t N, size_t M){
    s_matrix * p_matrix = (s_matrix *) malloc(sizeof(s_matrix));
    p_matrix->N = N;
    p_matrix->M = M;
    p_matrix->data = (double *) malloc(sizeof(double)*M*N);
    memset(p_matrix->data, 0, sizeof(double)*M*N);
    return p_matrix;
}
```

(d) Pour les deux implémentations possibles, écrivez la fonction pour libérer une matrice.

Solution: Avec un tabeau de tableaux :

```
void matrix_free(s_matrix * pm){
    free(pm->data[0]);
    free(pm->data);
    free(pm);
}
```

Avec un tableau de $N \times M$ éléments :

```
void matrix_free(s_matrix * pm){
    free(pm->data);
    free(pm);
}
```

(e) Pour les deux implémentations possibles, écrivez les fonctions pour lire et écrire dans la matrice.

Solution: Avec un tabeau de tableaux :

```
double matrix_get(s_matrix * pm, size_t i, size_t j){
    return pm->data[i][j];
}

void matrix_set(s_matrix * pm, size_t i, size_t j, double v){
    pm->data[i][j] = v;
}
```

Avec un tableau de $N \times M$ éléments :

```
double matrix_get(s_matrix * pm, size_t i, size_t j){
    return pm->data[i*(pm->M)+j];
}

void matrix_set(s_matrix * pm, size_t i, size_t j, double v){
```

```
6     pm->data[i*(pm->M)+j] = v;
7 }
```

(f) À partir des accesseur de la question (e), écrivez la fonction pour calculer le produit matriciel entre deux matrices. Pour rappel, la multiplication entre deux matrices $\mathbf{A} \in \mathcal{M}_{M,N}$ et $\mathbf{B} \in \mathcal{M}_{N,P}$ s'écrit :

$$c_{i,j} = \sum_{k=1}^{N} a_{i,k} \times b_{k,j} \tag{1}$$

avec $c_{i,j}$ la i-ème ligne et j-ème colonne de la matrice $\mathbf{C} \in \mathcal{M}_{M,P}$.

Solution:

```
1  s_matrix * matrix_prod(s_matrix * pmA, s_matrix * pmB){
2      s_matrix * pmC;
3      if(pmA->M != pmB->N)
4          return NULL;
5
6      pmC = matrix_alloc(pmA->N, pmB->M);
7      for(size_t i = 0 ; i < pmA->N ; i++)
8          for(size_t j = 0 ; j < pmB->M ; j++){
9              double c = 0.0;
10             for(size_t k = 0 ; k < pmA->M ; k++){
11                 double a = matrix_get(pmA, i, k);
12                 double b = matrix_get(pmB, k, j);
13                 c += a*b;
14             }
15         matrix_set(pmC, i, j, c);
16         }
17
18     return pmC;
19 }
```

(g) Quand l'on lit ou écrit dans la mémoire, il est important que les opérations successives soit faites sur des cases successives dans la mémoire. Il faut impérativement éviter les sautes dans la mémoire pour favoriser la mise en cache des données et diminué les temps d'accès aux données.
Dans votre réponse de la question (f), regardez si vous avec des sauts dans la mémoire et proposez des modifications dans vos codes pour éviter ces sauts en mémoire.

Solution: Dans la solution de la question (f), la ligne 11 du code génère des sauts dans la mémoire, en effet a chaque incrémentation de la variable k, l'on change de ligne dans la lecteur de la matrice $\mathbf{B}$, ce qui créé des discontinuité dans l'accès à la mémoire.

Pour éviter cela, il est possible d'inverser la boucle de la ligne 7 avec la boucle de la ligne 9 :

```
1  s_matrix * matrix_prod_v2(s_matrix * pmA, s_matrix * pmB){
2      s_matrix * pmC;
3      if(pmA->M != pmB->N)
4          return NULL;
5
6      pmC = matrix_alloc(pmA->N, pmB->M);
```

```
    for(size_t i = 0 ; i < pmA->N ; i++)
        for(size_t k = 0 ; k < pmA->M ; k++){
            double a = matrix_get(pmA, i, k);
            for(size_t j = 0 ; j < pmB->M ; j++){
                double b = matrix_get(pmB, k, j);
                matrix_set(pmC, i, j, matrix_get(pmC, i, j) + a*b);
            }
        }

    return pmC;
}
```

Voila un code de test pour évaluer le gain de performance entre les deux implémentions :

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>   // for gettimeofday()
#include <string.h>

struct struct_matrix {
    double * data;
    size_t N; // Nb lignes
    size_t M;// Nb Colonnes
};
typedef struct struct_matrix s_matrix;

s_matrix * matrix_alloc(size_t N, size_t M){
    s_matrix * p_matrix = (s_matrix *) malloc(sizeof(s_matrix));
    p_matrix->N = N;
    p_matrix->M = M;
    p_matrix->data = (double *) malloc(sizeof(double)*M*N);
    memset(p_matrix->data, 0, sizeof(double)*M*N);
    return p_matrix;
}

void matrix_free(s_matrix * pm){
    free(pm->data);
  free(pm);
}

double matrix_get(s_matrix * pm, size_t i, size_t j){
    return pm->data[i*(pm->M)+j];
}

void matrix_set(s_matrix * pm, size_t i, size_t j, double v){
    pm->data[i*(pm->M)+j] = v;
}

s_matrix * matrix_prod(s_matrix * pmA, s_matrix * pmB){
  s_matrix * pmC;
  if(pmA->M != pmB->N)
    return NULL;
  pmC = matrix_alloc(pmA->N, pmB->M);
  for(size_t i = 0 ; i < pmA->N ; i++)
    for(size_t j = 0 ; j < pmB->M ; j++){
```

```
43          double c = 0.0;
44        for(size_t k = 0 ; k < pmA->M ; k++){
45          double a = matrix_get(pmA, i, k);
46          double b = matrix_get(pmB, k, j);
47          c += a*b;
48        }
49        matrix_set(pmC, i, j, c);
50      }
51
52    return pmC;
53}
54
55s_matrix * matrix_prod_v2(s_matrix * pmA, s_matrix * pmB){
56    s_matrix * pmC;
57    if(pmA->M != pmB->N)
58      return NULL;
59
60    pmC = matrix_alloc(pmA->N, pmB->M);
61    for(size_t i = 0 ; i < pmA->N ; i++)
62      for(size_t k = 0 ; k < pmA->M ; k++){
63        double a = matrix_get(pmA, i, k);
64        for(size_t j = 0 ; j < pmB->M ; j++){
65          double b = matrix_get(pmB, k, j);
66          matrix_set(pmC, i, j, matrix_get(pmC, i, j) + a*b);
67        }
68      }
69
70    return pmC;
71}
72
73double rand_double(double a, double b){
74    return (rand()/((double) RAND_MAX))*(b-a)+b;
75}
76
77#define _N_ 2000
78
79int main(){
80    struct timeval start, end;
81
82      gettimeofday(&start, NULL);
83
84    srand(time(NULL));
85      s_matrix * A = matrix_alloc(_N_, _N_);
86      s_matrix * B = matrix_alloc(_N_, _N_);
87      for(size_t i = 0 ; i < _N_ ; i++)
88        for(size_t j = 0 ; j < _N_ ; j++){
89          matrix_set(A, i, j, rand_double(-1.0, 1.0));
90          matrix_set(B, i, j, rand_double(-1.0, 1.0));
91        }
92
93      gettimeofday(&start, NULL);
94      s_matrix * C = matrix_prod(A, B);
95      gettimeofday(&end, NULL);
96
```

```
    long micros = (((end.tv_sec - start.tv_sec) * 1000000) + end.tv_usec) - (start
    .tv_usec);

    printf("Implementassion classique : %lf seconde\n", micros/1e6f);

    gettimeofday(&start, NULL);
    C = matrix_prod_v2(A, B);
    gettimeofday(&end, NULL);

    micros = (((end.tv_sec - start.tv_sec) * 1000000) + end.tv_usec) - (start.
    tv_usec);

    printf("Implementassion opti. : %lf seconde\n", micros/1e6f);
    return 0;
}
```

À l'excution, nous obtenons le résultat suivant :

```
  $ gcc bench_exo1.c
  $ ./a.out
Implementassion classique : 75.347404 seconde
Implementassion opti. : 69.126846 seconde
  $
  $ gcc bench_exo1.c -O3
  $ ./a.out
Implementassion classique : 16.290503 seconde
Implementassion opti. : 5.397025 seconde
```

2. Exo. 2 - Recherche des k plus petits éléments d'un tableau `T`. Nous considérons T un tableau non trié de n entiers.
   (a) Écrivez un algorithme qui retourne les indexes des k plus petits éléments du tableau `T` en procédant par recherches successives.

Solution: En C :

```
#include <stdio.h>
#include <stdlib.h>

int is_in(size_t * tab, size_t n, size_t v){
  for(size_t i = 0 ; i < n ; i++)
    if(tab[i] == v)
      return 1;
  return 0;
}

void kpp(int * T, size_t n, size_t * index_tab, size_t K){
  for(size_t j = 0 ; j < K ; j++){
    for(size_t i = 0; i < n ; i++)
      if(!is_in(index_tab, j, i)){
        index_tab[j] = i;
        break;
      }
```

```c

    for(int i = 0 ; i < n ; i++){
      if(is_in(index_tab, j, i))
        continue;
      if(T[index_tab[j]]>T[i])
        index_tab[j] = i;
    }
  }
  return index_tab;
}

#define _K_ 4
int main(){
  int tab[] = {10, 3, 35, 5, 12, 86, 42, 5, 9, 1, 85};
  size_t index_tab[_K_];
  kpp(tab, 11, index_tab, _K_);
  for(size_t i = 0 ; i < _K_ ; i++)
    printf("%ld %d\n", index_tab[i], tab[index_tab[i]]);
  return 0;
}
```

Ou plus simplement en Python :

```python
def kpp(tab, k):
  index_kpp = set()

  for i in range(k):
    current_min_index = None

    for j in range(len(tab)):
      if j in index_kpp:
        continue
      if current_min_index is None:
        current_min_index = j;
        continue
      if tab[current_min_index] > tab[j]:
        current_min_index = j
    index_kpp.add(current_min_index)

  return index_kpp
```

(b) Écrivez un algorithme qui retourne les indexes des k plus petits éléments du tableau `T` en triant le tableau au préalable (la fonction de tri par ordre croissant est donnée `tri(T)` ; son coût est : $\mathcal{O}(n\log(n))$)

Solution: En C :

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct tuple{
  size_t idx;
  int val;
} s_tuple;
```

```c

int comp_tuple(s_tuple * a, s_tuple * b)
  return a->val - b->val;

size_t * sorted_kpp(int * T, size_t n, size_t * index_tab, size_t K){
  s_tuple * t_tuple = malloc(sizeof(s_tuple)*n);
  for(size_t i = 0 ; i < n ; i++){
    t_tuple[i].idx = i;
    t_tuple[i].val = T[i];
  }

  qsort(t_tuple, n, sizeof(s_tuple), (int(*)(const void *,const void *))comp_tuple
    );

  for(size_t k = 0 ; k < K ; k++)
    index_tab[k] = t_tuple[k].idx;

  free(t_tuple);
}

#define _K_ 4
int main(){
  int tab[] = {10, 3, 35, 5, 12, 86, 42, 5, 9, 1, 85};
  size_t index_tab[_K_];
  kpp(tab, 11, index_tab, _K_);
  for(size_t i = 0 ; i < _K_ ; i++)
    printf("%ld %d\n", index_tab[i], tab[index_tab[i]]);
  return 0;
}
```

Ou plus simplement en Python :

```python
def sorted_kpp(tab, k)
  sortedtab = sorted(enumerate(tab), key=lambda x:x[1])
  index_kpp = set()
  for i in range(k):
    index_kpp.add(sortedtab[i][0])
  return index_kpp;
```

(c) Comparez le temps mis par chacun des algorithmes pour $k = 1..n$.

Solution: Dans la question a, nous avons une boucle de n cycles imbriquée dans une boucle de K cycles. Intuitivement, nous pouvons en déduire que la complexité de cette algorithme est de l'ordre $\mathcal{O}(nK)$.

Dans la question b, nous avons une boucle de n cycles, suivie du tri, suivie d'une boucle de K cycles. La complexité majoritaire de ces trois partie étant la complexité du tri ($\mathcal{O}(n\log(n))$), l'algorithme est alors d'une complexité de l'ordre de $\mathcal{O}(n\log(n))$.

Nous pouvons en conclure que si K est plus petit que $\log(n)$, il est préférable d'utiliser l'algorithme de la question a, si non l'algorithme de la question b est préférable.