Correction du devoir surveillé

1. Files d'attente

1.1 Une liste de réels est soit la liste vide, notée ϕ, soit un couple constitué d'un réel (la tête de la liste) et d'une liste (la queue de la liste).

1.2 La représentation des listes en C suit leur définition mathématique : elle doit donc comporter 2 cas. On représentera les listes par des pointeurs sur une structure, le pointeur NULL représentant la liste vide. Pour une liste non vide, la structure pointée devra contenir la tête et la queue de la liste. Ainsi,

```
struct list_t {
    head : float ;
    tail : struct list_t * ;
}
```

Par commodité, on utilisera l'alias suivant :

```
typedef struct list_t*   list ;
```

1.3

```
list cons(float h, list t) {
    list r = (list) malloc(sizeof(struct list_t));
    r -> head = h;
    r -> tail = t;
    return r;
}
```

1.4 (a) opération : sum

type : $\mathbb{L} \to \mathbb{R}$

description : calcule la somme des éléments présents dans une liste de réels.

définition : $\begin{cases} sum(\emptyset) = 0 \\ sum(h::t) = h + sum(t) \end{cases}$

où $\mathbb{L}$ représente l'ensemble des listes de nombres réels et $h::t$ est la liste de tête h et de queue t.

(b)
```
float sum(list l) {
    if(l == NULL) return 0;
    else return l->head + sum(l->tail);
}
```

(c)
```
float sum(list l) {
    float r = 0;
    while (l != NULL) {
        r += l->head;
        l = l->tail;
    }
    return r;
}
```

1.5 L'opération get n'est applicable à une liste que si celle-ci est non-vide

```
float get(list l) {
    assert(l != NULL);
    return l->head;
}
```

1.6

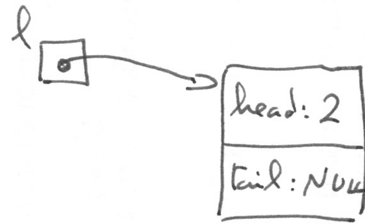
```
list pop(list l) {
    if (l == NULL) return l;
    else { list r = l -> tail;
        free(l);
        return r;
    }
}
```

1.7

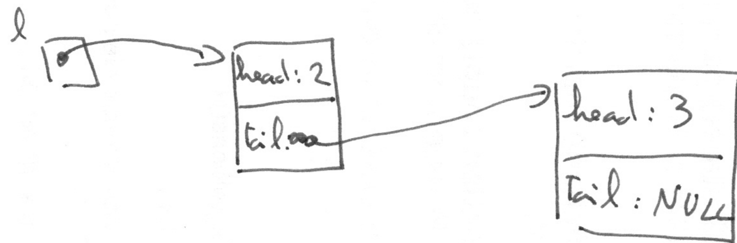
```
list push(float e, list l) {
    if(l == NULL) return cons(e, l);
    else {
        l -> tail = push(e, l -> tail);
        return l;
    }
}
```

1.8

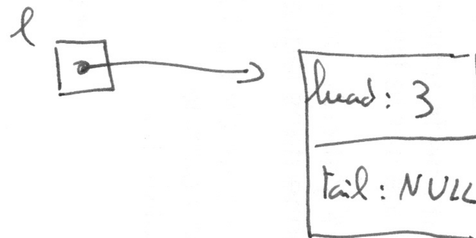
list l = push(2, NULL)



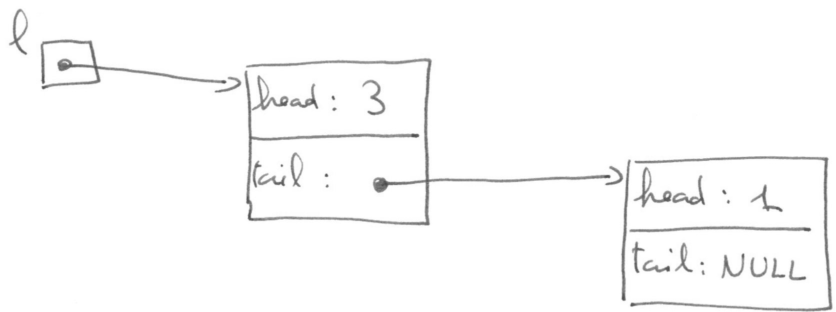
l = push(3, l)



l = pop(l)



l = push(1, l)



---

l = pop(l)

Le programme affiche successivement 2, 3 et 1.

1.9 Une liste équipée des opérations get, push et pop représente le fonctionnement abstrait d'une file d'attente :

- on entre à la fin de la file (push)
- le premier à en sortir est le premier élément (get, pop).

Si chaque élément de la liste correspond à une durée, la fonction sum calcule la durée totale d'attente dans la file (utile par exemple pour savoir si on veut ou non rentrer dans la file).

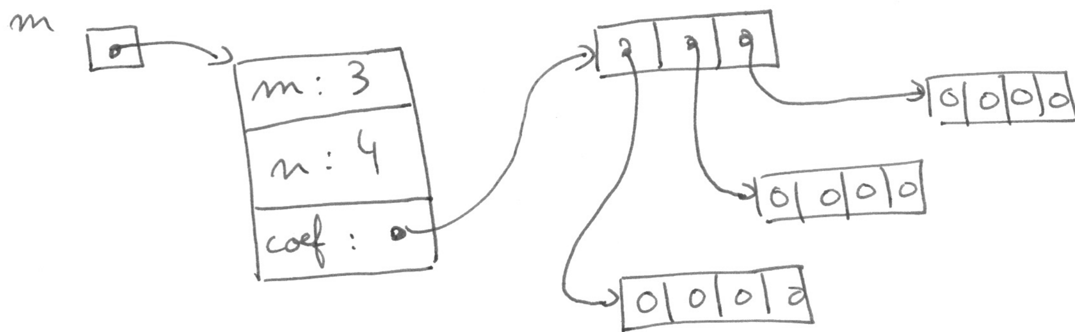
2. Représentation mémoire des matrices

2.1

```
matrix create_matrix(int m, int n) {
    matrix r = (matrix) malloc(sizeof(struct matrix_t));          1 alloc
    r->m = m;  r->n = n;
    r->coef = (float**) malloc(m * sizeof(float*));               1 alloc
    for (int i = 0; i < m; i++) {
        r->coef[i] = (float*) malloc(n * sizeof(float));          1 alloc    boucle
        for (int j = 0; j < n; j++) r->coef[i][j] = 0;                       m itérations
    }
    return r;
}
```

Comme légendé dans le code, cette fonction produit bien $m+2$ allocations.

2.2 matrix m = create_matrix (3,4)



2.3 Si la matrice est symétrique, on peut se contenter de stocker la matrice triangulaire supérieure, ou inférieure :

```
matrix create_matrix (int m) {
    matrix r = (matrix) malloc(sizeof(struct matrix_t));
    r->m = m;   r->n = m;
    r->coef = (float**) malloc (m* sizeof(float*));
    for (int i = 0; i < m; i++) {
        r->coef[i] = (float*) malloc ((i+1) * sizeof(float));
        for (int j = 0; j <= i; j++) r->coef[i][j] = 0;
    }
    return r;
}
```

Note : comme la matrice est symétrique, elle est aussi carrée, d'où la simplification du prototype. L'espace mémoire occupé par une matrice se décompose comme suit

structure : 3 champs de 4 octets chacun
tableau des pointeurs : $m \times 4$ octets

coefficients :

- $m \times n \times 4$ octets dans le cas général
- $\frac{m(m+1)}{2} \times 4$ octets dans le cas symétrique

$$\frac{m(m+1)}{2} = \sum_{i=1}^{m} i$$

soit

$$4(3+m+m.n) \quad \text{et} \quad 4\left(3+m+\frac{m(m+1)}{2}\right)$$ octets respectivement.

2.4

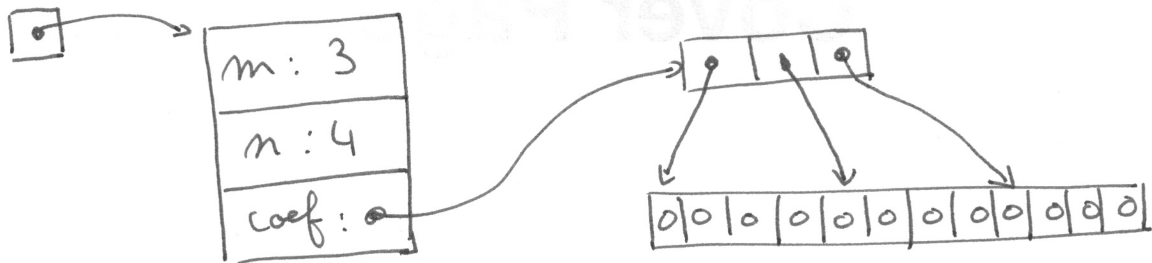
```
float get(int i, int j, matrix m) {
    assert(i >= 0); assert(j >= 0);
    assert(i < m->m); assert(j < m->n);
    if (i < j) return m->coef[j][i];
    else return m->coef[i][j];
}
```

2.5

```
~~matrix create_matrix(int m, int n) {
    matrix r = (matrix) malloc(sizeof(struct matrix_t));
    r->m = m; r->n = n;
    r->coef = (float**) malloc(m*n*sizeof(float));~~
```

```
matrix create_matrix(int m, int n) {
    matrix r = (matrix) malloc(sizeof(struct matrix_t));
    float* coef = (float*) malloc(m*n* sizeof(float));
    r->m = m;   r->n = n;
    r->coef = (float**) malloc(m*sizeof(float*));
    for (int i = 0; i < m; i++) {
        r->coef[i] = coef + i*n;
        for (int j = 0; j < n; j++) r->coef[i][j] = 0;
    }
    return r; }
```

2.6 `matrix m = create_matrix(3, 4);`



Les coefficients sont effectivement regroupés en mémoire.

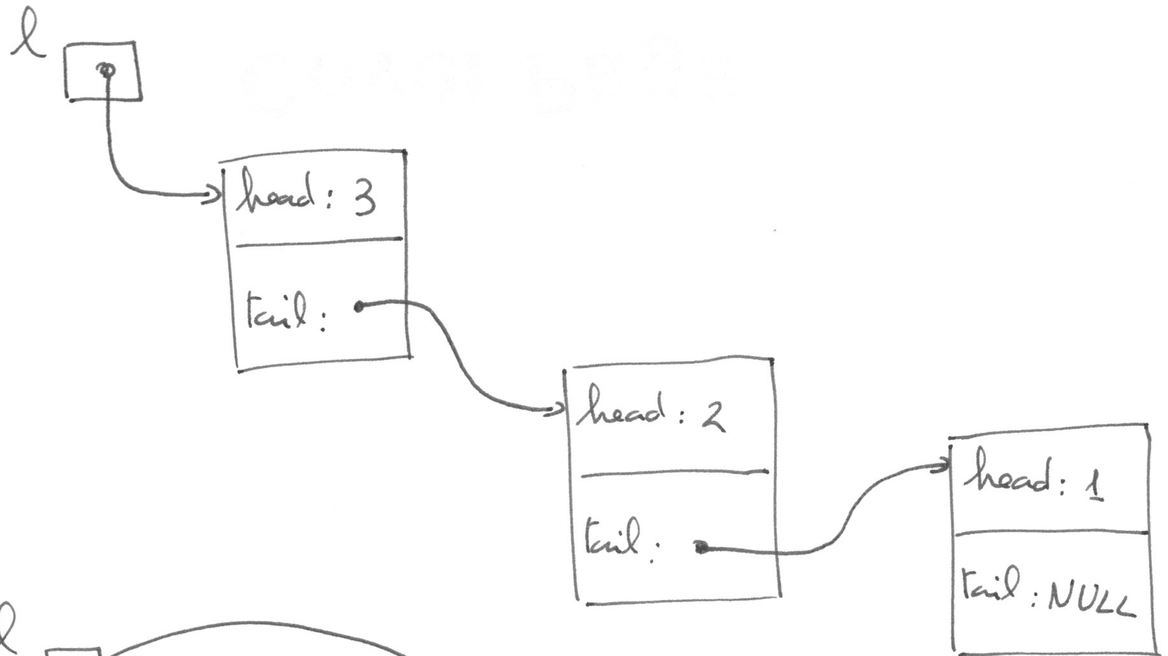
3. Tri par décantation

3.1
```
int has_inversion(list l) {
    return (l != NULL) &&
           (l->tail != NULL) &&
           (l->head > l->tail->head);
}
```

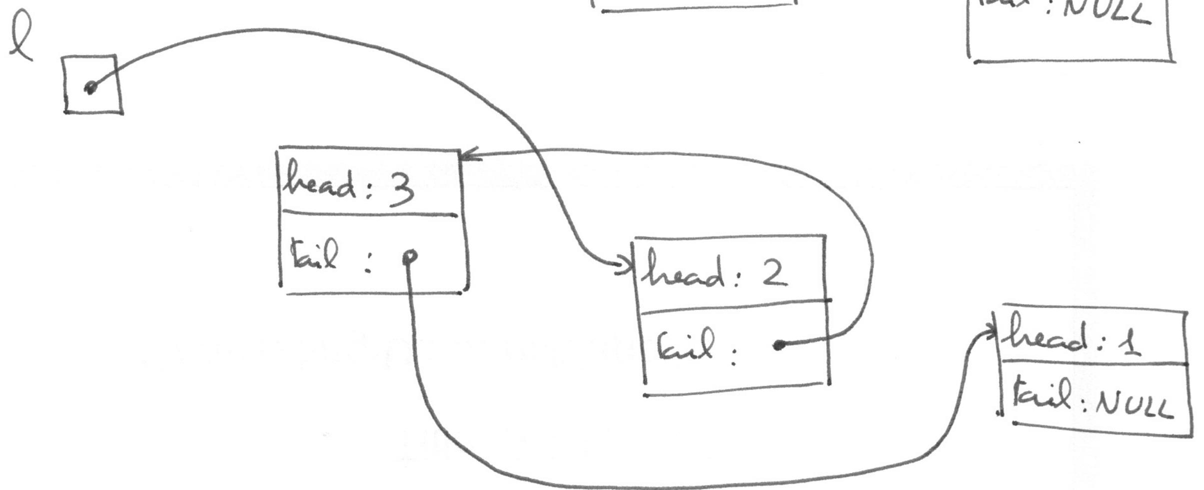
3.2
```
list swap_head(list l) {
    assert(l != NULL && l->tail != NULL);
    list r = l->tail;
    list rest = l->tail->tail;
    r->tail = l
    r->tail->tail = rest;
    return r;
}
```

3.3

avant :



après :



3.4
```
list decantation (list l) {
    if (has_inversion (l)) {
        l = swap_head (l);
        l->tail = decantation (l->tail);
    }
    return l;
}
```

3.5 Soit $T(n)$ la complexité (au pire) de decantation pour une liste de taille n

has_inversion est en temps constant, tout comme swap-head

On a donc au pire $T(n) = k + T(n-1)$ et $T(0) = k$

Ce type de récurrence admet une solution linéaire en n

3.6

```
list sort (list l) {
    if (l == NULL) return l;
    l->tail = sort (l->tail);
    return decantation (l);
}
```

3.7 Soit $U(n)$ la complexité de sort pour les listes de longueur n.

On a $U(n) = U(n-1) + T(n)$ et $U(0) = 0$

Intuitivement $T(n) \simeq n$

d'où $U(n) \simeq n + (n-1) + (n-2) + \dots + 0$

sort est donc de complexité quadratique.

Epilogue : calcul de complexité "sérieux" (enfin un peu plus...)

On cherche à estimer la complexité en nombre de comparaisons au pire

$$T_{has_inversion}(n) = \begin{cases} 0 & \text{pour } n = 0, 1 \\ 1 & \end{cases}$$

$$T_{swap_head}(n) = 0$$

Pour décantation, le pire cas est obtenu lorsqu'il y a une inversion à chaque étape (liste triée en ordre décroissant).

$$T_{décantation}(n) = \begin{cases} 0 = T_{has_inversion}(n) & \text{pour } n = 0, 1 \\ \underbrace{T_{has_inversion}(n)}_{=1} + T_{décantation}(n-1) & \end{cases}$$

$$T_{décantation}(n) = \begin{cases} 0 \text{ pour } n = 0, 1 \\ n-1 \text{ sinon} \end{cases}$$

$$T_{sort}(n) = \begin{cases} 0 & \text{si } n = 0 \\ T_{sort}(n-1) + T_{décantation}(n) & \text{sinon} \end{cases}$$

$$= \begin{cases} 0 & \text{si } n = 0 \\ T_{sort}(n-1) + n - 1 = \sum_{n=1}^{n} n-1 \end{cases}$$

$$= \frac{n(n-1)}{2}$$

$\leadsto$ complexité quadratique.