

Calculatrice et documents de cours interdits. Tout le code doit être écrit en C.

Il y a entre  $-0.5$  et  $0.5$  points pour le soin/clarté.

## I Petites questions

- (/0.5) 1. Écrire une fonction `abs` de prototype `double abs(double)` permettant de calculer la valeur absolue d'un `double`.

Solution :

```
1  double abs(double x) {  
2      if(x < 0.)  
3          return -x;  
4      return x;  
5  }
```

- (/1) 2. Écrire une fonction `swap` de prototype `void swap(int*, int*)` échangeant les valeurs de deux variables (données par leurs adresses).

Solution :

```
1  void swap(int* x, int* y) {  
2      int tmp = *x;  
3      *x = *y;  
4      *y = tmp;  
5  }
```

- (/0.5) 3. Écrire des instructions pour définir deux variables de type `int` puis échanger leurs valeurs avec `swap`.

Solution :

```
1  int x = 1;  
2  int y = 2;  
3  swap(&x, &y);
```

- (/1) 4. Écrire une fonction `premier` de prototype `bool premier(int)` déterminant si un entier est premier.

Solution :

```
1  bool premier(int n) {  
2      for(int i = 2; i < n; i++) {  
3          if(n % i == 0)  
4              return false;  
5      }  
6      return true;  
7  }
```

## II Fractions

- (/0.5) 1. Définir une structure `fraction` contenant un numérateur et un dénominateur (deux `int`).

Solution :

```
1  typedef struct fraction {  
2      int num;  
3      int den;  
4  } fraction;
```

(/0.5) 2. Définir une variable de type `fraction` correspondant à la fraction  $\frac{3}{2}$ .

**Solution :**

```
1  fraction f;  
2  f.num = 3;  
3  f.den = 2;
```

(/1) 3. Écrire une fonction `add` pour ajouter deux fractions, de prototype `fraction add(fraction, fraction)`.  
**Remarque :** Il n'est pas nécessaire d'utiliser de pointeur/malloc.

**Solution :**

```
1  fraction add(fraction f1, fraction f2) {  
2      fraction res;  
3      res.num = f1.num*f2.den + f1.den*f2.num;  
4      res.den = f1.den*f2.den;  
5      return res;  
6  }
```

### III Mélange de Knuth

Dans cet exercice, tous les tableaux contiennent des entiers (`int`).

Le mélange de Knuth est un algorithme qui applique une permutation aléatoire sur un tableau (il mélange aléatoirement les éléments). Pour l'implémenter on va utiliser la fonction `rand` en C qui permet d'obtenir un entier aléatoire entre 0 et `UINT_MAX`, c'est-à-dire le plus grand entier non signé représentable.

(/0.5) 1. En supposant les entiers non signé codés sur 4 octets, quelle est la valeur de `UINT_MAX`?

**Solution :** 4 octets = 64 bits. Le plus grand entier positif correspond à  $\underbrace{1\dots1}_{32}_2 = 2^{32} - 1$ .

(/1) 2. Écrire une fonction `randn`, telle que, si `n` est un entier positif, `randn(n)` renvoie un entier aléatoire entre 0 et `n`. (Remarque : il n'a pas besoin d'être exactement uniformément aléatoire).

**Solution :** La solution ci-dessous est préférable à une boucle `while` qui pourrait faire un nombre arbitraire d'opérations.

```
1  int randn(int n) {  
2      return rand() % (n+1);  
3  }
```

(/0.5) 3. Écrire une fonction `swap` telle que, si `t` est un tableau d'entiers et `i, j` deux de ses indices, `swap(t, i, j)` échange les éléments d'indice `i` et `j` dans `t`.

**Solution :**

```
1  void swap(int* t, int i, int j) {  
2      int tmp = t[i];  
3      t[i] = t[j];  
4      t[j] = tmp;  
5  }
```

Voici le pseudo-code du mélange de Knuth sur un tableau `t` de taille `n` :

Pour `i` variant de 0 à `n - 1`:

Soit `j` un entier aléatoire entre 0 et `i`

Echanger les éléments d'indice `i` et `j` dans `t`

- (/0.5) 4. Traduire le mélange de Knuth en C, telle que `shuffle(t, n)` applique le mélange de Knuth sur un tableau `t` d'entiers de taille `n`.

**Solution :**

```
1 void shuffle(int* t, int n) {  
2     for(int i = 0; i < n; i++)  
3         swap(t, i, randn(i));  
4 }
```

## IV Tri par insertion (dichotomique)

Dans cet exercice, tous les tableaux contiennent des entiers (`int`).

- (/0.5) 1. Écrire une fonction `position` telle que, si `t` est un tableau trié d'entiers et `n, e` des entiers, `position(t, n, e)` renvoie le plus petit indice `i < n` tel que `t[i] > e`. Si un tel indice n'existe pas, on renverra `n`.  
Par exemple, si `t` contient les éléments 1, 3, 6, 9, 17 (dans cet ordre), `position(t, 4, 7)` doit renvoyer 3.  
On demande une complexité en  $O(n)$ .

**Solution :**

```
1 int position(int* t, int n, int e) {  
2     for(int i = 0; i < n; i++)  
3         if(t[i] > e)  
4             return i;  
5     return n;  
6 }
```

- (/2.5) 2. Réécrire la fonction précédente en s'inspirant de la recherche par dichotomie. Quelle est la nouvelle complexité?

**Solution :** En adaptant la preuve du cours, on pourrait montrer que la fonction suivante termine et renvoie la bonne valeur.

```
1 int position_dicho(int* t, int n, int e) {  
2     int i = 0, j = n - 1;  
3     while(i < j) {  
4         int m = (i + j)/2;  
5         if(t[m] < e) {  
6             i = m + 1;  
7         }  
8         else  
9             j = m; // et non pas m - 1, car il est possible que la valeur à renvoyer soit m  
10    }  
11    return i;  
12 }
```

- (/1) 3. Écrire une fonction `decaler` telle que `decaler(t, i, j)` décale les éléments du tableau `t` d'une position vers la droite. Ainsi, la valeur de `t[i]` doit être mise dans `t[i + 1]`, `t[i + 1]` dans `t[i + 2]`, ..., `t[j - 1]` doit être mise dans `t[j]`. Par exemple, après les instructions `int t[] = {1, 3, 6, 9, 17}` et `decaler(t, 1, 3)`, `t` doit contenir 1, 3, 3, 6, 17 (`t[i]` n'est pas modifié).

**Solution :**

```
1 void decaler(int* t, int i, int j) {  
2     for(; j > i; j--)  
3         t[j] = t[j - 1];  
4 }
```

Le tri par insertion sur un tableau `t` consiste à parcourir chaque élément `t[i]` de `t` et à l'insérer à sa bonne position de façon à

ce que les  $i$  premiers éléments soient triés. On a donc une boucle **for** avec l'invariant suivant : « au  $i$ ème passage de la boucle **for**, les  $i$  premiers éléments de **t** sont triés ».

- (/1) 4. En réutilisant **position** et **decaler**, implémenter une fonction **tri** permettant d'effectuer le tri par insertion sur un tableau. **tri** aura donc le prototype `void tri(int* t, int n)`, où **n** est la taille de **t**.

**Solution :**

```
1 void tri(int* t, int n) {
2     for(int i = 0; i < n; i++) {
3         int p = position(t, i, t[i]);
4         int tmp = t[i];
5         decaler(t, p, i);
6         t[p] = tmp;
7     }
8 }
```

- (/0.5) 5. Quelle est la complexité de **tri** en utilisant la fonction de la question 1?

**Solution :** On passe  $n$  fois dans le **for**, en appelant **position** et **decaler** en  $O(n)$ , d'où  $O(n^2)$  au total.

- (/1) 6. Quelle est la complexité de **tri** en utilisant la fonction de la question 2? Et si on compte seulement le nombre de comparaisons (c'est-à-dire le nombre de tests de  $<$ ,  $>$ ,  $==$ ...)?

**Solution :** Même si on utilise **position\_dicho**, le fait qu'on utilise  $n$  fois la fonction **decaler** fait que la complexité totale reste  $O(n^2)$ .

Par contre, si on compte seulement le nombre de comparaisons, on obtient  $O(n \log(n))$ .

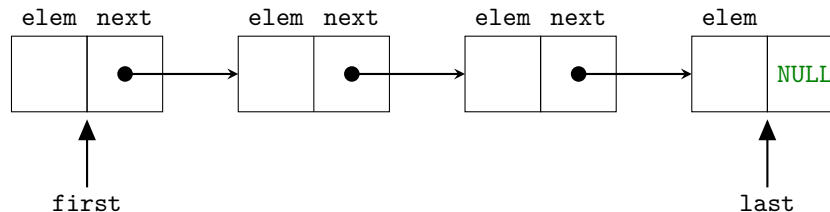
## V Implémentation d'une file par liste chaînée

Dans cet exercice, on utilise une liste chaînée d'entiers (type **list**) pour implémenter une file (type **file**). Une file contient un pointeur vers le 1er élément et vers le dernier.

```
1 typedef struct list {
2     list* next;
3     int elem;
4 } list;
```

```
1 typedef struct file {
2     list* first;
3     list* last;
4 } file;
```

Voici un exemple de **file** :



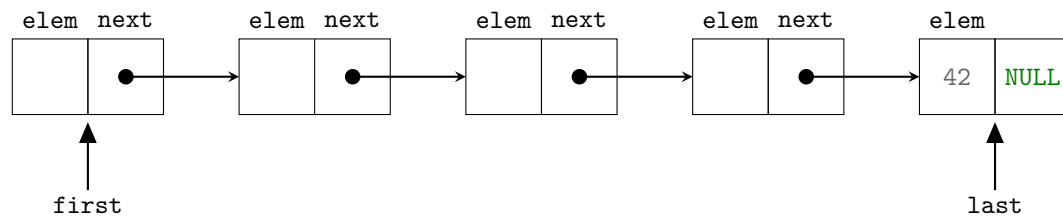
Par convention, la **file** vide possède deux pointeurs **first** et **last** qui sont **NULL**.

- (/0.5) 1. Écrire une fonction pour tester si une **file** est vide.

**Solution :**

```
1 bool empty(file* f) {
2     return f->first == NULL && f->last == NULL;
3 }
```

- (/1.5) 2. Écrire une fonction de prototype `void add(file*, int)` pour ajouter un élément à la fin de la file (après `last`). Ainsi, `add(f, 42)` appliqué sur la file `f` de l'exemple ci-dessus doit modifier la file de la façon suivante :



Solution :

```

1 void add(file* f, int e) {
2     list* noeud = malloc(sizeof(list));
3     noeud->elem = e;
4     noeud->next = NULL;
5     if(f->last == NULL) { // si la file est vide
6         f->last = noeud;
7         f->first = noeud;
8     }
9     else {
10        f->last->next = noeud;
11        f->last = noeud;
12    }
13 }
```

- (/1.5) 3. Écrire une fonction de prototype `int pop(file*)` pour supprimer et renvoyer l'élément au début de la file. On fera attention à libérer la mémoire.

Solution :

```

1 int pop(file* f) {
2     int e = file->first->elem;
3     list* second = file->first->next;
4     free(file->first);
5     file->first = second;
6     return e;
7 }
```

## VI Fusion de listes triées

- (/2) 1. Écrire une fonction `list* merge(list*, list*)` (le type `list` étant défini ci-dessus) telle que, si `l1` et `l2` sont deux listes triées (par ordre croissant), `merge(l1, l2)` renvoie une liste triée contenant les éléments des deux listes.

**Solution :** Il n'est pas spécifié si on peut modifier les listes en arguments. Voici une solution, en les modifiant :

```

1 list* merge(list* l1, list* l2) {
2     if(l1 == NULL)
3         return l2;
4     if(l2 == NULL)
5         return l1;
6     if(l2->elem < l1->elem)
7         return merge(l2, l1);
8     l1->next = merge(l1->next, l2);
9     return l1;
10 }
```