DS 3 d'informatique corrigé
MP2I, 3 heures

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

I Petites questions

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

> **Solution** :

```c
double abs(double x) {
    if(x < 0.)
        return -x;
    return x;
}
```

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

> **Solution** :

```c
void swap(int* x, int* y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

3. Écrire des instructions pour définir deux variables de type `int` puis échanger leurs valeurs avec `swap`.

> **Solution** :

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

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

> **Solution** :

```c
bool premier(int n) {
    for(int i = 2; i < n; i++) {
        if(n % i == 0)
            return false;
    }
    return true;
}
```

II Fractions

1. Définir une structure `fraction` contenant un numérateur et un numérateur (deux `int`).

> **Solution** :

```c
struct fraction {
    int num;
    int den;
}
```

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

Solution :

```c
struct fraction f;
f.num = 3;
f.den = 2;
```

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 :

```c
fraction add(fraction f1, fraction f2) {
    fraction res;
    res.num = f1.num*f2.den + f1.den*f2.num;
    res.den = f1.den*f2.den;
    return res;
}
```

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.

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 $\underbrace{1\ldots1}_{32}{}_2 = 2^{32} - 1$.

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 :

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

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 :

```c
void swap(int* t, int i, int j) {
    int tmp = t[i];
    t[i] = t[j];
    t[j] = tmp;
}
```

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
```

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 :

```c
void suffle(int* t, int n) {
    for(int i = 0; i < n, i++)
        swap(t, i, randn(i));
}
```

IV Tri par insertion (dichotomique)

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

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 2.
On demande une complexité en $O(n)$.

Solution :

```c
int position(int* t, int n, int e) {
    for(int i = 0; i < n; i++)
        if(t[i] > e)
            return i;
    return n;
}
```

2. Réécrire la fonction précédente en s'inspirant de la recherche par dichotomie. Quelle est la nouvelle complexité?

Solution :

```c
int position_dicho(int* t, int n, int e) {
    int i = 0;
    int j = n - 1;
    while(i < j) {
        int m = (i + j)/2;
        if(t[m] < e)
            j = m;
        else
            i = m + 1;
    }
    return i;
}
```

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 :

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

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.

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 :

```c
void tri(int* t, int n) {
    for(int i = 0; i < n; i++) {
        int p = position(t, i, t[i]);
        int tmp = t[i];
        decaler(t, p, i);
        t[p] = tmp;
    }
}
```

5. Quelle est la complexité de `tri` en utilisant la fonction de la question 1?

Solution :

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 <, >, ==...)?

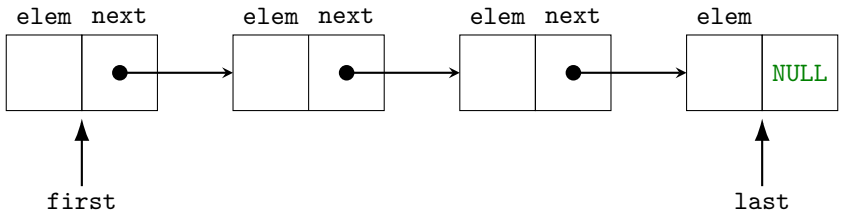
V Implémentation d'une file par liste chaînée

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.

```c
typedef struct list list;
struct list {
    list* next;
    int elem;
};
```

```c
struct file {
    list* first;
    list* last;
};
```

Voici un exemple de `file` :



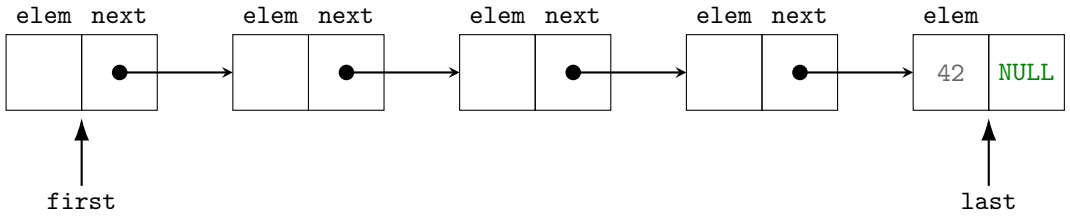
Par convention, la `file` vide possède deux pointeurs `first` et `last` qui sont `NULL`.

1. Écrire une fonction pour tester si une `file` est vide.

Solution :

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

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 :

```
void add(file* f, int e) {
    list* noeud = malloc(sizeof(list));
    noeud->elem = e;
    noeud->next = NULL;
    f->last->next = noeud;
    f->last = noeud;
}
```

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 :

```
int pop(file* f) {
    int e = file->first->elem;
    list* second = file->first->next;
    free(file->first);
    file->first = second;
    return e;
}
```

VI Fusion de listes triées

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 :

```
list* merge(list* l1, list* l2) {
    if(l1 == NULL)
        return l2;
    if(l2 == NULL)
        return l1;
    if(l1->elem < l2->elem) {
        list* l = merge(l1->next, l2);
        l1->next = l;
        return l1;
    }
    list* l = merge(l1, l2->next);
    l2->next = l;
    return l2;
}
```