TD : Le langage C

1 Premiers pas

Types et opérations élémentaires

Exercice 1 : Propriétés du Xor

On note $\mathbb{U} := \{0, 1\}$ et on définit l'opération « exclusive or », notée `^` et appelée aussi Xor, par

a \ b	0	1
0	0	1
1	1	0

a `^` b

Soit $N \in \mathbb{N}$. On considère l'ensemble $\mathbb{U}^N$ des nombres binaires non signés sur N bits et on note `^` la loi de composition interne Xor, définie terme à terme sur $\mathbb{U}^N$.

1. La loi `^` est-elle commutative ? Est-elle associative ?
2. La loi `^` a-t-elle un élément neutre ?
3. Si $x \in \mathbb{U}^N$, que vaut `x ^ x` ?
4. Que peut-on dire de $(\mathbb{U}^N, \text{`^`})$?
5. Supposons qu'Alice et Bob veuillent communiquer de manière chiffrée un message de N bits et qu'ils disposent d'une suite de N bits, partagée, qu'ils sont les seuls à connaître. Comment peuvent-ils procéder ?

Ce procédé d'encryption est appelé one-time pad et a été utilisé pour la première fois par Frank Miller en 1882.

Exercice 2 : Extraction de bits

On considère un entier non signé

$$n = \overline{b_{N-1} \dots b_1 b_0}^2$$

codé sur N bits où $N = 64$, c'est-à-dire un élément de `uint64_t`. Dans les fonctions que vous proposerez, seules les constante 0 et `const uint64_t one = 1` ainsi que les opérateurs `&`, `<<`, `>>`, `-` et `==` seront autorisés. Notamment, les opérateurs `*`, `/` et `%` sont interdits.

1. Écrire une fonction `bool est_pair(uint64_t n)` déterminant si n est pair.
2. Écrire une fonction `uint64_t fill(int k)` renvoyant l'entier dont la représentation binaire est $\overline{1 \dots 1}^2$, formée de k bits égaux à 1, avec $k \leqslant N$?
3. Écrire une fonction `bool est_divisible(uint64_t n, int k)` déterminant si l'entier n est divisible par 2^k.
4. Écrire une fonction `uint64_t quotient(uint64_t n, int k)` donnant le quotient de la division euclidienne de n par 2^k. Écrire une fonction `uint64_t reste(uint64_t n, int k)` donnant son reste.
5. Écrire une fonction `uint64_t extrait(uint64_t n, int k)` permettant d'extraire le k-ème bit de n.
6. Écrire une fonction `uint64_t extrait_zone(uint64_t n, int k, int l)` permettant d'extraire les bits de poids $2^k, \dots, 2^{k+l-1}$ de n où $0 \leqslant l \leqslant N - k$.

Les opérateurs `&`, `<<`, `>>`, `-` et `==` ne nécessitent qu'un seul cycle processeur pour s'exécuter. Ils sont donc beaucoup plus efficaces que les opérateurs `/` et `%` qui s'exécutent chacun en une vingtaine de cycles processeur. Les fonctions que nous venons d'écrire sont donc extrêmement efficaces.

Exercice 3 : Popcount

On considère toujours un entier n non signé sur N bits. On prendra $N = 64$ ce qui permettra de travailler avec le type `uint64_t`.

1. Écrire une fonction `bool est_puissance_deux(uint64_t n)` déterminant si n est une puissance de 2 en effectuant un nombre d'opérations indépendant de N. Seules les opérateurs `&`, `-` et `==` sont autorisées dans cette fonction.

2. On définit popcount(n) comme le nombre de bits égaux à 1 dans la représentation de n. Comment calculer popcount(n) en utilisant un nombre d'opérations proportionnel à popcount(n), et non à N ?
3. Écrire une fonction `popcount` implémentant cet algorithme et ayant le prototype suivant :

```c
int popcount(uint64_t n);
```

2 Pointeur, tableau et structure

Pointeur

Exercice 4 : Exercice

On considère la fonction suivante :

```c
void mystere(int *x, int *y) {
    *x = *x - *y;
    *y = *x + *y;
    *x = *y - *x;
}
```

1. Quel affichage obtiendrait-on avec le code suivant ?

```c
int x = 3;
int y = 4;
mystere(&x, &y);
printf("x = %d\n", x);
printf("y = %d\n", y);
```

2. De manière générale, quel est l'effet de la fonction `mystere` ? On justifiera.
3. Quel affichage obtiendrait-on avec le code suivant ?

```c
int x2 = 3;
int y2 = 3;
mystere(&x2, &y2);
printf("x2 = %d\n", x2);
printf("y2 = %d\n", y2);
```

4. Quel affichage obtiendrait-on avec le code suivant ?

```c
int x3 = 3;
mystere(&x3, &x3);
printf("x3 = %d\n", x3);
```

Quel est le problème dans la démonstration faite plus haut ?

Exercice 5 : Exercice

1. Écrire une fonction `incremente` qui prend en entrée un pointeur vers un entier et incrémente la valeur de cet entier de 1. Cette fonction ne renverra rien.
2. Dans tous les exemples suivants, on souhaite qu'un appel `f(px, py)` incrémente celle des valeurs pointées par `px` et `py` qui est la plus petite, ou celle pointée par `px` en cas d'égalité. Par exemple :

```c
#include <stdio.h>

int main() {
    int x = 4;
    int y = 3;
    printf("x = %d, y = %d\n", x, y);
    f(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    f(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

doit donner l'affichage suivant :

```
x = 4, y = 3
x = 4, y = 4
x = 5, y = 4
```

Dans chaque cas, dire si :
— Il y a un problème de type, et si oui, où.
— Les fonctions ont bien le comportement attendu (uniquement dans les cas où il n'y a pas de problème de type). Si ce n'est pas le cas, on expliquera d'où vient le problème.

(a) Code 1

```c
int plus_petit(int x, int y) {
    if (x <= y) return x;
    return y;
}

void f(int *px, int *py) {
    incremente(plus_petit(*px, *py));
}
```

(b) Code 2

```c
int *plus_petit(int x, int y) {
    if (x <= y) return &x;
    return &y;
}

void f(int *px, int *py) {
    incremente(plus_petit(*px, *py));
}
```

(c) Code 3

```c
int *plus_petit(int *x, int *y) {
    if (*x <= *y) return x;
    return y;
}

void f(int *px, int *py){
    incremente(plus_petit(px, py));
}
```

(d) Code 4

```
int *plus_petit(int *x, int *y) {
    int a = *x;
    int b = *y;
    if (a <= b) return &a;
    return &b;
}

void f(int *px, int *py){
    incremente(plus_petit(px, py));
}
```

Tableau

Exercice 6 : Exercice

Écrire une fonction `bool is_sorted(int a[], int n)` qui renvoie `true` si et seulement si le tableau `a` de taille `n` est trié dans l'ordre croissant.

Solution. Voici une solution :

```
bool is_sorted(int a[], int n) {
    for (int i = 0; i < n - 1; i++) {
        if (a[i] > a[i + 1]) {
            return false;
        }
    }
    return true;
}
```

□

Exercice 7 : Swap

Écrire une fonction `void swap(int a[], int i, int j)` qui échange les éléments d'indices `i` et `j` du tableau `a`. On fait l'hypothèse que les entiers `i` et `j` désignent bien des indices valides du tableau `a`.

Solution. Voici une solution :

```
void swap(int a[], int i, int j) {
    int v = a[i];
    a[i] = a[j];
    a[j] = v;
}
```

□

Exercice 8 : Mélange de Fisher-Yates

Pour mélanger les éléments d'un tableau aléatoirement, il existe un algorithme très simple qui procède ainsi : on parcourt le tableau de gauche vers la droite et, pour chaque élément à l'indice i, on l'échange avec un élément situé à un indice tiré aléatoirement entre 0 et i inclus. Cet algorithme s'appelle le mélange de Knuth ou encore le mélange de Fisher-Yates. Écrire une fonction `void knuth_shuffle(int a[], int n)` qui réalise cet algorithme pour un tableau `a` de taille `n`. On rappelle qu'on tire un entier aléatoire entre 0 et $n-1$ inclus avec `rand() % n`. On pourra se resservir de la fonction `swap` de l'exercice précédent.

Exercice 9 : Exercice

1. Écrire une fonction `void two_way_sort(int a[], int n)` qui trie en place un tableau `a` qui contient uniquement les valeurs 0 et 1, en n'effectuant que des échanges dans le tableau avec la fonction `swap` de l'exercice donné plus haut. La complexité doit être proportionnelle au nombre n d'éléments.
2. Donner un invariant prouvant la correction de votre algorithme.

Solution. Voici une solution :

```
void two_way_sort(int a[], int n) {
    int i = 0;
    int j = 0;
    while (j < n) {
        if (a[j] == 0) {
            swap(a, i, j);
        }
        j++;
    }
}
```

L'invariant est le fait que les $a[k]$ pour $0 \leqslant k < i$ sont égaux à 0 et les $a[k]$ pour $i \leqslant k < j$ sont égaux à 1. □

Exercice 10 : Drapeau hollandais

Écrire une fonction `void dutch_flag(int a[], int n)` qui trie en place un tableau `a` qui contient uniquement les valeurs 0, 1 et 2, en n'effectuant que des échanges dans le tableau avec la fonction `swap` de l'exercice donné plus haut. La complexité doit être proportionnelle au nombre n d'éléments.

On appelle cela le problème du drapeau hollandais, *car il a été initialement présenté par W.H.J. Feijen avec un tableau contenant les trois couleurs du drapeau hollandais (rouge, blanc, bleu). Le problème a été popularisé par E.W. Dijkstra, lui-même néerlandais.*

Exercice 11 : Exercice

Écrire une fonction `void insertion_sort(int a[], int n)` qui trie le tableau `a` de taille `n` avec un tri par insertion. Le principe est de parcourir le tableau de la gauche vers la droite, en maintenant une partie déjà triée sur la gauche, et d'insérer l'élément suivant v dans la partie déjà triée. Pour cela, on décale vers la droite les éléments déjà triés tant qu'ils sont plus grands que v. Écrire le code avec deux boucles imbriquées.

Exercice 12 : Recherche dichotomique et dépassement de capacité

1. Écrire une fonction `int binary_search(int v, int a[], int n)` qui cherche une occurence de la valeur `v` dans un tableau `a` de taille `n` supposé trié par ordre croissant, à l'aide d'une recherche dichotomique. Si `v` apparaît dans `a`, la fonction renvoie un indice où `v` apparait. Sinon, elle renvoie `-1`.
2. Cette fonction risque-t-elle un débordement de capacité ? Si oui, proposez une solution pour le résoudre.

Exercice 13 : Exercice

On veut tester la fonction `binary_search` de l'exercice précédent. Proposez une façon de procéder.

Exercice 14 : La bourse

On cherche à calculer le gain maximum possible à la Bourse sur une action pendant une période de n jours, en ne faisant qu'une opération d'achat et de vente d'une seule action. On suppose que les cours quotidiens de cette action sont enregistrés dans un tableau `t` d'entiers de longeur n. On définit l'amplitude de la variation du cours comme la valeur absolue du maximum de la variation de ce cours pendant la période observée, c'est-à-dire la quantité suivante :

$$\text{amplitude} = \max_{0 \leqslant i \leqslant j < n} |t_j - t_i| = \max_{0 \leqslant i < n} t_i - \min_{0 \leqslant i < n} t_i.$$

1. Écrire une fonction `int amplitude(int t[], int n)` qui renvoie comme résultat l'amplitude de la variation du cours. Quelle est la complexité temporelle de cette fonction ?

Le gain maximum est le gain maximum possible sur la période observée, c'est-à-dire la quantité

$$\text{gain} = \max_{0 \leqslant i \leqslant j < n} (t_j - t_i).$$

2. Donner un exemple où l'amplitude est différente du gain maximum.
3. En suivant textuellement la définition du gain, écrire une fonction

   `int gain(int t[], int n, int *achat, int *vente)`

   qui renvoie le gain maximal possible et donne des dates d'achat et de vente réalisant un tel gain sur une période de durée minimale.

Pour tout j $(0 \leqslant j < n)$ définissons le gain courant maximum gainCourant_j comme le gain maximum possible obtenu en vendant son action au temps j, c'est-à-dire

$$\text{gainCourant}_j = \max_{0 \leqslant i \leqslant j} (t_j - t_i).$$

4. En calculant progressivement le gain courant maximum, écrire une fonction

   `int gain_lineaire(int t[], int n, int *achat, int *vente)`

   qui renvoie, en temps linéaire par rapport à n, le gain maximum possible et donne des dates d'achat et de vente correspondant sur une période de durée minimale.

On considère maintenant la possibilité de faire séquentiellement deux transactions pendant la période observée, c'est-à-dire de considérer deux dates d'achat i et i', et deux dates de vente j et j' telles que $0 \leqslant i \leqslant j \leqslant i' \leqslant j' < n$.

5. Écrire une fonction

   `int gain_deux_actions(int t[], int n, int *achat1, int *vente1, int *achat2, int *vente2)`

   qui renvoie, en temps quadratique par rapport à n, le gain maximum possible en faisant deux transactions sur le cours de l'action donnant les quatres dates i, j, et i', j' d'achats et de ventes de l'action permettant d'obtenir le gain maximum, avec $(j - i) + (j' - i')$ minimum. Quelle est la complexité de cet algorithme ?

Exercice 15 : Exercice

Dans cet exercice, on se propose de programmer le *tri rapide*, un algorithme de tri inventé par C.A.R. Hoare en 1961, et de le faire comme il faut. On cherche à écrire une fonction `void quicksort(int a[], int n)` pour trier en place un tableau `a` de `n` entiers. Les idées de cet algorithme sont les suivantes :

- On commence par écrire une fonction plus générale, qui trie le segment du tableau `a` compris entre l'indice `l` inclus et l'indice `r` exclus, avec le profil `void quickrec(int a[], int l, int r)`.
- Pour trier le segment `a[l..r[`, on procède ainsi :
  - On choisit une valeur `p` dans ce segment appelé *pivot*. On peut prendre par exemple la valeur `a[l]`.
  - En utilisant des échanges, on réorganise les éléments du segment `a[l..r[` avec sur la plage `[l..lo[` les éléments $< p$, puis sur la plage `[lo..hi[` les éléments égaux à p puis sur la plage `[hi..r[` les éléments $> p$. C'est la fonction que l'on demande d'écrire en premier lieu.
  - On trie récursivement les deux segments `a[l..lo[` et `a[hi..r[` avec la fonction `quickrec`.
- Enfin, pour trier le tableau entier, on commence par le mélanger avec l'algorithme de Knuth de l'exercice donné plus haut, puis on appelle la fonction `quickrec` sur l'intégralité du tableau. Ainsi, le choix de la valeur pivot est randomisé.

Écrivez le code des fonctions `quickrec` et `quicksort`.

Exercice 16 : Bit Scan Reverse

Pour un entier $n = \overline{b_{N-1} \dots b_0}^2$ non nul et non signé codé sur N bits, on définit $\text{bsr}(n)$, pour « bit scan reverse », comme le plus grand $i \in [\![0, N[\![$ tel que $x_i = 1$.

1. Proposer une fonction simple de prototype

   ```
   int bsr(uint64_t n);
   ```

2. On suppose qu'on a précalculé un tableau `int t_bsr[256]` tel que `t_bsr[i]` contienne $\text{bsr}(i)$ pour $0 \leqslant i < 256$. Ré-écrire la fonction `bsr` pour qu'elle effectue au maximum trois décalages.

Exercice 17 : Palindrome

On va travailler sur la fonction suivante, qui vérifie qu'une chaîne de caractères est bien un palindrome :

```
bool palindrome(char* s, int n) {
    int debut = 0;
    int fin = n - 1;
    while (debut < fin) {
        if (s[debut] != s[fin]) return false;
        debut = debut + 1;
        fin = fin - 1;
    }
    return true;
}
```

1. Écrire la spécification de cette fonction.
2. Démontrer sa terminaison, puis sa correction.

Exercice 18 : Coefficient binomial

On va travailler sur la fonction suivante calculant $\binom{n}{k}$ pour $n \in \mathbb{N}$ et $k \in [\![0, n]\!]$.

```
int binome(int k, int n) {
    int *ligne = malloc(sizeof(int) * (n + 1));
    for (int i = 0; i <= n; i++) {
        ligne[i] = 1;
        for (int j = i - 1; j > 0; j--) {
            ligne[j] = ligne[j] + ligne[j - 1];
        }
    }
    return ligne[k];
}
```

1. Cette fonction présente un comportement non désiré. Quelle correction faut-il lui apporter ?
2. Énoncer un invariant de boucle pour la boucle extérieure.

Exercice 19 : Tri par dénombrement

Le tri par dénombrement est une méthode de tri très efficace quand on trie des entiers positifs dont on connaît un majorant m qui n'est pas « trop grand » par rapport au nombre n d'entiers à trier. Le principe est, pour chaque entier $k \in [\![0, m]\!]$, de compter le nombre d'occurences de k dans le tableau à trier, puis d'en déduire le tableau trié.

1. Écrire une fonction `int nombre_occurences(int t[], int n, int k)` qui calcule le nombre d'occurences de `k` dans le tableau `t` de longueur n.
2. En déduire la fonction `int* tableau_occurences(int t[], int n, int m)` qui renvoie un tableau de taille $m + 1$ qui contient pour tout indice k le nombre d'occurences de k dans le tableau t.
3. Déterminer la complexité temporelle et spatiale de `tableau_occurences`.
4. Implémenter une fonction `void tri_denombrement(int t[], int n, int m)` qui trie *en place* le tableau.
5. Quelle est sa complexité temporelle et spatiale ? On l'exprimera en fonction de la taille n du tableau et de m.
6. Améliorer la complexité de vos fonctions.