Devoir Surveillé n°1

Informatique MP2I

Durée : **2 heures**

Tous documents, calculatrices, machines connectées interdits.

Ce sujet est composé de 4 exercices indépendants. Ce sujet comporte 5 pages.

Les programmes doivent être écrits en C. Une grande importance sera apportée à la clarté et à la lisibilité. On pourra toujours librement utiliser une fonction démandée dans une question précédente, même si cette question n'a pas été traitée. On suppose que les bibliothèques `stdbool.h`, `stdio.h`, `stdlib.h` et `assert.h` ont déjà été importées et que vous n'avez donc pas à le faire.

Exercice 1 Entiers négabinaires

Le système négabinaire (aussi appelé base -2) est une méthode alternative de représentation des entiers relatifs. Certains ordinateurs expérimentaux polonais ont utilisé cette représentation dans les années 50.
On dispose de deux chiffres, 0 et 1. La suite de chiffres $b_k b_{k-1} \dots b_1 b_0$ représente le nombre :

$$\sum_{i=0}^{k} b_i \times (-2)^i$$

Pour indiquer qu'une suite de chiffres $b_k b_{k-1} \dots b_1 b_0$ doit être interprétée à l'aide du système négabinaire, on utilisera la notation $\overline{b_k b_{k-1} \dots b_1 b_0}^{-2}$.

Q1. Quelle est la représentation décimale de $\overline{11010}^{-2}$? Quelle est la représentation décimale de $\overline{111010}^{-2}$?

On propose de représenter les entiers négabinaires en C par des tableaux d'entiers dont les seuls éléments sont des `1` et des `0`. Chaque chiffre b_i sera stocké dans la case en position `i` du tableau.

Q2. Écrire une fonction `bool verifie_legal(int t[], int taille)` qui renvoie `true` si et seulement si le tableau `t` de taille `taille` ne contient que des `0` et des `1`.

Q3. Écrire une fonction de tests permettant de vérifier le bon comportement de la fonction `verifie_legal`.

Q4. Écrire une fonction `int negabin_to_deci(int t[], int taille)` qui renvoie l'entier représenté par le tableau `t` de taille `taille` selon le système négabinaire. Vous pourrez supposer que le tableau `t` ne contient effectivement que des `1` et des `0`.

Q5a. Montrer que pour tout $n \in \mathbb{N}^*$, $2^n = (\sum_{i=0}^{n-1} 2^i) + 1$.

Q5b. Soit $N = \overline{b_k b_{k-1} \dots b_1 b_0}^{-2}$ un nombre négabinaire tel que $b_k = 1$. Montrer que N est négatif si et seulement si k est impair.

Exercice 2 Automates cellulaires élémentaires

Un **automate cellulaire élémentaire** est un procédé permettant de faire évoluer une séquence binaire à l'aide d'une **règle**. Ces séquences binaires sont de taille impaire, et seront représentées avec une case blanche pour un 0 et une case noire pour un 1. En C, elles seront implémentées par un tableau d'entiers, chaque entier valant 0 ou 1. La séquence initiale $S^{(0)}$ est nommée **graine**. Elle ne contient que des 0, à l'exception de la valeur centrale qui vaut 1 :



Q1. Écrire une fonction de prototype `int *graine(int n)` qui alloue dynamiquement un tableau de n entiers, les initialise pour représenter une graine puis renvoie le pointeur associé. On ne vérifiera pas que n est bien impair.

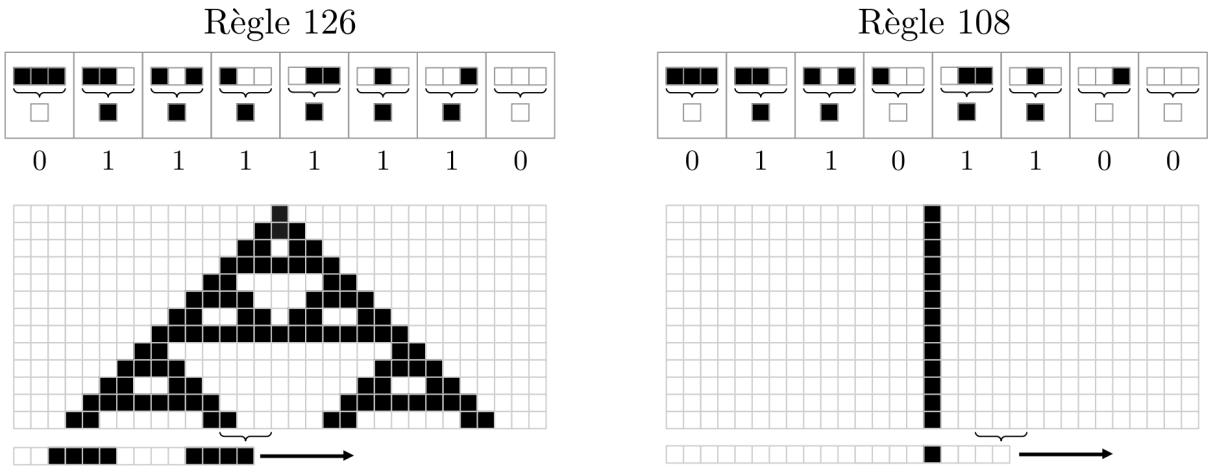
À partir d'une séquence binaire $S^{(k)} = b_0^{(k)} \dots b_{n-1}^{(k)}$, l'automate calcule la séquence suivante $S^{(k+1)} = b_0^{(k+1)} \dots b_{n-1}^{(k+1)}$ (de même taille) en suivant sa règle R. Dans cette nouvelle séquence, chaque case est obtenue en appliquant la règle R sur trois cases de la séquence précédente : celle de même indice, celle à gauche et celle à droite.

$$b_i^{(k+1)} = R\left(b_{i-1}^{(k)}, b_i^{(k)}, b_{i+1}^{(k)}\right)$$

Dans le cas où $b_{i-1}^{(k)}$ ou $b_{i+1}^{(k)}$ ne sont pas définis car les indices dépassent les limites, on leur donnera la valeur 0.

Q2. Justifier qu'il existe exactement 256 règles différentes.

L'objectif est de suivre une règle pour visualiser l'évolution d'une graine, en représentant les séquences obtenues ligne par ligne comme illustré ci-dessous. Des motifs plus ou moins complexes peuvent émerger selon la règle choisie.



Une règle sera désignée par un entier r entre 0 et 255. On définit cet entier r à l'aide de sa décomposition binaire $b_7 b_6 \dots b_0$, le bit de poids fort b_7 étant à gauche. Pour chaque indice i entre 0 et 7, b_i est le résultat de la règle R appliquée aux trois bits de la décomposition binaire de i (bit de poids fort à gauche). Par exemple, si l'on souhaite connaître b_3, on commence par décomposer 3 en binaire ce qui donne les trois valeurs 0, 1 et 1. La valeur de b_3 est alors $R(0, 1, 1)$.

L'exemple ci-dessus illustre le détail des règles 126 et 108, en explicitant ce lien avec la représentation binaire des deux nombres.

Q3. À l'image de ce qui a été fait pour la règle 126 et 108 dans l'exemple, détailler ce que renvoie la règle 110 sur les 8 configurations possibles. Représenter une graine de taille 7, puis les trois lignes suivantes obtenues en appliquant la règle 110.

Q4. Écrire une fonction `int bit_i(int n, int i)` qui renvoie le $i$ème bit de l'encodage binaire de l'entier n, le bit d'indice 0 étant celui de poids faible.

Q5. Écrire une fonction `int applique_regle(int gauche, int milieu, int droite, int regle)` qui à partir des trois valeurs de cases $b_{i-1}^{(k)}, b_i^{(k)}, b_{i+1}^{(k)}$ et d'un numéro de règle, renvoie la bonne valeur de sortie $R\left(b_{i-1}^{(k)}, b_i^{(k)}, b_{i+1}^{(k)}\right)$ de l'automate.

Q6. Écrire une fonction `sequence_suivante` de prototype bien choisi qui prend un numéro de règle, une séquence binaire $S^{(k)}$ et sa taille, qui alloue dynamiquement une nouvelle sequence de même taille et la remplit pour représenter la séquence suivante $S^{(k+1)}$ avant de renvoyer le pointeur associé.

Q7. Écrire une fonction de prototype `void affiche_automate(int regle, int n, int nb_lignes)` qui affiche les premières `nb_lignes` obtenues à partir d'une graine de taille n en suivant la règle donnée. On pourra afficher un croisillon (#) pour une case noire et un espace pour une case blanche.

Exercice 3 Aliasing d'un alphabet

Q1. Représenter le plus fidèlement possible l'état de la mémoire lorsque l'exécution du programme suivant atteint le drapeau **pour la première fois**.

```c
void affiche_code(char* t[], int taille) {
    // 🚩
    for (int i=0; i<taille; i++) {
        printf("%c", *t[i]);
    }
    printf("...\n");
}

int main() {

    char a[] = {'S', 'I', 'T', 'E'};

    char* t1[] = {&a[0], &a[1], &a[3], &a[0], &a[2], &a[3], &a[0]};


    affiche_code(t1, 7);

    a[0] = a[2];
    a[1] = a[3];
    a[2] = 'A';
    a[3] = 'N';

    affiche_code(t1, 7);

    return 0;
}
```

Q2. Donner l'affichage produit par ce programme.

Exercice 4 Codage par plage

Dans cet exercice on se propose d'appliquer le **codage par plage** (ou *run-length encoding*) à des tableaux d'entiers. Le codage par plage est une transformation sur un tableau t qui convertit chaque série d'éléments identiques consécutifs de t en une paire d'entiers : le premier désigne la longueur de la série, et le second l'élément qui y est répété.

Par exemple le tableau `t` contenant les éléments `1, 1, 1, 1, 0, 8, 8, 8, 8, 8, 1` est encodé par la séquence suivante : `4, 1, 1, 0, 5, 8, 1, 1`. En effet `t` contient tout d'abord quatre fois l'élément `1` (d'où `4, 1`), puis une fois l'élément `0` (d'où `1, 0`), puis cinq fois l'élément `8` (d'où `5, 8`) et enfin une fois l'élément `1` (d'où `1, 1`).

Q1. Quel est le codage par plage du tableau dont les éléments sont `2, 2, 2, 6, 5, 5, 2, 2, 2, 2, 2`?

On souhaite écrire une fonction qui calcule l'encodage par plage d'un tableau `t` donné en entrée et stocke cet encodage dans un nouveau tableau.

Q2. Justifier l'intérêt d'une allocation dynamique sur le tas dans cette situation, en expliquant pourquoi un stockage sur la pile n'est pas approprié.

Q3. Écrire une fonction `int rle_length(int t[], int taille)` qui renvoie la taille attendue de la séquence obtenue par codage par plage du tableau `t` de taille `taille`.

Q4. Écrire une fonction `int* tab_to_rle(int t[], int taille)` qui applique le codage par plage au tableau `t` de taille `taille`. On renverra la séquence obtenue dans un nouveau tableau.

Le codage par plage est un codage **sans pertes** : à partir de la séquence codée, on peut retrouver le tableau originel. Les entiers aux positions paires indiquent un nombre de répétitions, et l'entier suivant (en position impaire) indique l'élément à répéter.

Par exemple, `4, 1, 1, 0, 5, 8, 1, 1` se décode ainsi : répéter 4 fois l'élément `1`, puis répéter 1 fois l'élément `0`, puis répéter 5 fois l'élément `8`, puis répéter 1 fois l'élément `1`. On retrouve la séquence `1, 1, 1, 1, 0, 8, 8, 8, 8, 8, 1` qui correspond bien au tableau originel.

Q5. Décoder la séquence `1, 3, 1, 1, 2, 2, 2, 1`

Q6. Écrire une fonction `int tab_length(int s[], int taille)` qui prend en entrée un tableau `s` de taille `taille` obtenu par codage par plage, et qui renvoie la taille attendue de la séquence obtenue par décodage de `s`.

Q7. Écrire une fonction `int* rle_to_tab(int s[], int taille)` qui prend en entrée un tableau `s` de taille `taille` obtenu par codage par plage, et qui applique l'algorithme de décodage pour reconstruire le tableau originel. On renverra un pointeur vers ce tableau reconstruit.

Q8. Expliquer comment on pourrait modifier cette fonction pour récupérer à la fois le nouveau tableau et sa taille.