Nom:

Prénom:

Matricule:

Complément d'informatique

Examen écrit, 20 janvier 2020

Livre ouvert. Durée : 2h30.

Notes importantes :

- *Les feuilles d'énoncé ne peuvent pas être désagrafées et doivent être toutes rendues avec vos noms, prénoms et matricules sur la première page (celle-ci).*
- *Les questions 1, 3.1, 3.2, et 6 doivent être répondues directement sur les feuilles d'énoncé. Les autres questions doivent être répondues sur des feuilles* **séparées** *(par question, pas par sous-question), sur lesquelles doivent figurer vos nom, prénom et matricule. Si vous n'avez pas répondu à une question, merci de remettre néanmoins une feuille blanche à votre nom ou de préciser dans l'énoncé que vous n'avez pas répondu à cette question.*
- *L'examen est à livre ouvert : les documents autorisés sont les slides du cours théorique, ainsi que les slides de la partie théorique du cours INFO2009.*

Question 1 (complexité)

Pour les fonctions suivantes, donnez leur complexité en temps dans le pire cas en notation O, en fonction de l'argument n.

```
void mystery1(int *tab, int s, int n) {
    int d = 0;
    if (s < 0) {
        d = n - (-s % n);
    } else {
        d = s % n;
    }
    int *new_tab = malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        new_tab[(i + d) % n] = tab[i];
    }
    for (int i = 0; i <n; i++)
        tab[i] = new_tab[i];
    free(new_tab);
}

void mystery2(int *tab, int n)
{
    int i, loc, j, selected;

    for (i = 1; i < n; ++i)
    {
        j = i - 1;
        selected = tab[i];

        loc = binarySearch(tab, selected, 0, j);

        while (j >= loc)
        {
            tab[j+1] = tab[j];
            j--;
        }
        tab[j+1] = selected;
    }
}
```

```
void mystery3(int *tab,int n) {
  int i = 0;
  int j = n-1;
  if (j>i) {
    while (j>i) {
      int p = rand() % 2;
      if (p==1) {
           int tmp = tab[i];
           tab[i] = tab[j];
           tab[j] = tmp;
      }
      j = j - 1;
      i = i + 1;
    }
    mystery3(tab, j+1);
    mystery3(tab+j+1, n-j-1);
  }
}
```

mystery1

mystery2

mystery3

Question 2 (algorithmique)

1. Soit deux tableaux `s` et `e` de longueurs `n` contenant respectivement les débuts et fins de n intervalles de temps. Écrivez une fonction `float union_length(float *s, float *e, int n)` calculant la longueur totale de l'union de ces intervalles. Par exemple si les deux tableaux contiennent les valeurs suivantes :

   $$\texttt{s} = [2, 1, 7, 6], \texttt{e} = [4.5, 3, 8, 9],$$

   correspondants aux quatre intervalles $\{[2; 4.5], [1; 3], [7, 8], [6; 9]\}$ la longueur totale renvoyée par la fonction devra être 6.5 (l'union de $[1; 3]$ et de $[2; 4.5]$ est en effet l'intervalle $[1; 4.5]$ de longueur 3.5 et l'union des intervalles $[7; 8]$ et $[6; 9]$ est l'intervalle $[6; 9]$ de longueur 3, qui n'a pas de recouvrement avec l'intervalle $[1; 4.5]$).
2. Donnez la complexité de votre solution.

Remarque : vous pouvez réutiliser toutes les fonctions vues au cours (en les modifiant si nécessaire).

Question 3 (arbre)

Soit l'implémentation d'arbre binaire vue au cours (slides 330 et 334 à 336) :

1. Donnez la séquence de valeurs affichée par le code suivant :

```
BTree *tree = btCreate();
BTNode *n1 = btCreateRoot(tree, 6);
BTNode *n2 = btInsertLeft(tree, n1, 2);
BTNode *n3 = btInsertLeft(tree, n2, 1);
BTNode *n4 = btInsertRight(tree, n2, 4);
BTNode *n5 = btInsertRight(tree, n1, 8);
BTNode *n6 = btInsertRight(tree, n5, 11);
btBreadthFirstTreeWalk(tree);
```

2. Quel parcours devriez-vous utiliser à la dernière ligne pour obtenir la séquence suivante ?
   `6 2 1 4 8 11`
3. En supposant que les nœuds de l'arbre sont décorés par des valeurs entières positives, écrivez une fonction `int tree_max_at_depth(BTree *tree, int k)` renvoyant la valeur maximale présente aux noeuds de profondeur k dans l'arbre, 0 s'il n'y a aucun nœud à cette profondeur. Vous pouvez supposer que cette fonction sera implémentée dans le fichier `BTree.c`.

Question 4 (généricité)

Soit un dictionnaire dont l'interface est donnée dans le fichier d'entête `dico.h` suivant :

```
typedef struct Dict_t Dict;

Dict *dictCreate(void);
void  dictFree(Dict *d);
void  dictInsert(Dict *d, char *key, void *value);
int   dictSearch(Dict *d, char *key);
```

En supposant que le dictionnaire est implémenté sur base d'une liste liée, avec la structure `Dict_t` définie de la manière suivante (dans le fichier `dico.c` associé) :

```
typedef struct Node_t {
  char *key;
  void *value;
  struct Node_t *next;
} Node;

struct Dict_t {
  Node *first;
  unsigned int nKeys;
};
```

On aimerait ajouter à l'implémentation une fonction `dicoMap(dict, f)` permettant d'appliquer une fonction $f(key, value)$ à toutes les paires `(key,value)` du dictionnaire. La fonction `f` ainsi que la fonction `dicoMap` ne doivent renvoyer aucune valeur.

1. Proposez une implémentation de cette fonction (en en déterminant la signature complète, c'est-à-dire le type des arguments et de la sortie)
2. Montrez comment appeler `dicoMap` pour afficher toutes les paires de clés-valeurs présentes dans le dictionnaire si on suppose que les valeurs associées aux clés sont des chaînes de caractères.

Question 5 (dictionnaire)

Soit une structure d'ensemble générique telle que définie par le fichier `.h` ci-dessous :

```
typedef struct Set_t Set;
Set   *setCreate();
void   setFree(Set *s);
void   setInsert(Set *s, char *key, int key_length);
int    setSearch(Set *s, char *key, int key_length);
```

et permettant de stocker des chaînes de caractères (l'argument `key_length` fourni aux fonctions représente la longueur de la chaîne `key` qui ne devra donc pas nécessairement se terminer par un caractère nul) :

1. En vous basant sur cette structure, implémentez une fonction `int hasRepetition(char *sequence, int n, int m)` renvoyant 1 si la chaîne de caractères `sequence` (de longueur `n`) contient au moins une sous-séquence répétée de longueur `m` (en supposant $\texttt{m} < \texttt{n}$), 0 sinon.
2. Donnez la complexité de cette fonction dans le pire cas en notation grand-O pour les trois implémentations du dictionnaire vues au cours :
   (a) Liste liée
   (b) Vecteur trié
   (c) Table de hachage

Question 6 (maîtrise du C et débogage)

Le code de la page suivante définit une grille de démineur et les fonctions de création et de libération de mémoire associées, comme dans le cadre du projet 1. Ce code contient plusieurs erreurs qui peuvent créer des problèmes à la compilation, à l'utilisation, ou encore simplement violer certains principes de bonne programmation. Indiquez toutes les erreurs que vous trouvez le plus précisément possible et corrigez les directement sur la feuille d'énoncé.

Remarque : pour simplifier le code, la gestion d'un retour NULL de la fonction `malloc` a été ignorée. Cela ne doit pas être considéré comme une erreur.

—- Grid.h —-

```
typedef struct Grid_t Grid;
typedef struct Cell_t Cell;

Grid *gridInit(int width, int height, int nbrBombs);
void  gridFree(Grid *grid);
```

—- Grid.c —-

```
struct Cell_t {
  int explored;
  int bomb;
  int flagged;
};

struct Grid_t {
  Cell **g;
  int width;
  int height;
  int nbrBombs;
};

Grid *gridInit(int width, int height, int nbrBombs) {
  Grid *grid = malloc(sizeof(Grid));

  grid->width = width;
  grid->height = height;
  grid->nbrBombs = nbrBombs;

  grid->g = malloc(width * sizeof(Cell **));
  for (int i = 0; i < width; i++) {
    grid->g[i] = malloc(height * sizeof(Cell *));
    for (int j = 0; j < height; j++) {
      grid->g[i][j] = malloc(sizeof(Cell));
      grid->g[i][j].bomb = 0;
      grid->g[i][j].explored = 0;
      grid->g[i][j].flagged = 0;
    }
  }

  return g;
}

void gridFree(Grid *grid) {
    for (int i = 0; i < grid->width; i++) {
        for (int j = 0; j < grid->height; j++) {
            free(grid->g[i][j]);
        }
    }
    free(grid->g);
    free(grid);
    return;
}
```