Nom:

Prénom:

Matricule:

Complément d'informatique

Examen écrit, 10 janvier 2019

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, 2, 4 et 6 (seulement le point 6.1) 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 (récursivité)

Soit le programme suivant :

```
#include <stdio.h>

int recur(int count, int n) {
  int newCount = count;
  if (n >= 1) {
    newCount = recur(newCount, n-1);              // Line A
    newCount++; printf("%d%d ", newCount, n);     // Line B
    newCount = recur(newCount, n-1);              // Line C
  }
  return newCount;
}

int main() {
  recur(0, 3);
}
```

Que va afficher le programme ?

1. Tel qu'il est écrit.

2. Si on inverse les lignes A et B.

3. Quelle est la complexité en temps, en notation grand-O, de la fonction `recur` en fonction de son argument `n` ?

Question 2 (analyse de complexité)

1. Donnez les complexités en temps des fonctions `mystery1` et `mystery2` ci-dessous en fonction de la taille `n` du tableau donné en argument. Dans le cas de `mystery2`, on supposera que le tableau ne contient que des valeurs 0 ou 1.

```
int mystery1(int a[], int n) {
  int max = -1;
  for (int k = 0; k < n-3; k++) {
     int sum = 0;
     for (int j = k; j < k+3; j++) {
       sum += a[j];
     }
     if (sum > max) max = sum;
  }
  return max;
}

void mystery2(int a[], int n) {
  int i = 0;
  int j = n-1;

  while (i<j) {
    if (tab[i] == 0)
      i++;
    else if (tab[j] == 1)
      j--;
    else {
      int tmp = tab[i];
      tab[i] = tab[j];
      tab[j] = tmp;
    }
  }
}
```

Mystery1 :

Mystery2 :

2. Soit la nouvelle heuristique ci-dessous pour résoudre le problème du voyageur de commerce (qui modifie le tour passé en argument), suivant l'interface du projet 2 :

```
static TourPosition *findClosest(Tour *tour, Town *town) {
  double minDist = DBL_MAX; // le plus grand double représentable en C
  TourPosition *minPos = NULL;
  TourPosition *currentPos = getTourStartPosition(tour);

  while (currentPos != NULL) {
    double d = distanceBetweenTowns(town, getTownAtPosition(tour, currentPos));
    if (d < minDist) {
      minPos = currentPos;
      minDist = d;
    }
    currentPos = getNextTourPosition(tour, currentPos);
  }
  return minPos;
}

Tour *heuristic3(Tour *tour) {
  Tour *newTour = createEmptyTour();

  Town *currentTown = getTownAtPosition(tour, getTourStartPosition(tour));
  addTownAtTourEnd(newTour, currentTown);
  removeTourPosition(tour, getTourStartPosition(tour));

  while (getTourSize(tour) != 0) {
    TourPosition *closestPos = findClosest(tour, currentTown);
    currentTown = getTownAtPosition(tour, closestPos);
    addTownAtTourEnd(newTour, currentTown);
    removeTourPosition(tour, closestPos);
  }
  return newTour;
}
```

La fonction `removeTourPosition(tour,pos)` supprime la position `pos` dans `tour`. En supposant que tous les appels aux fonctions de manipulation de tour sont $O(1)$, excepté l'appel à la nouvelle fonction `removeTourPosition` qui est $O(n)$ pour un tour de longueur n, donnez la complexité en temps au pire cas de cette heuristique en fonction de la taille n de `tour`.

Question 3 (algorithmique)

Répondez sur une feuille séparée à cette question.

Un tableau *trié décalé* ("rotated" en anglais) est défini comme un tableau trié dont on a décalé les éléments circulairement d'un nombre arbitraire (potentiellement nul) de positions. Par exemple, les tableaux $[3, 4, 5, 6, 1, 2]$ ou $[6, 1, 2, 3, 4, 5]$ sont de tableaux triés décalés obtenus du tableau trié $[1, 2, 3, 4, 5, 6]$ (qui est lui-même également un tableau trié décalé).

1. Écrivez une fonction `int minRotated(int tab[], int n)` de complexité strictement inférieure à $O(n)$ renvoyant la position du minimum dans un tableau `tab` trié décalé de longueur `n`. Par exemple, sur le tableau $[3, 4, 5, 6, 1, 2]$, cette fonction devra renvoyer 4. Précisez la complexité de cette fonction.
2. En vous servant de cette fonction, écrivez une fonction `int searchRotated(int key, int tab[], int n)`, également de complexité strictement inférieure à $O(n)$, renvoyant la position de la clé `key` dans le tableau trié décalé `tab` de longueur `n` ou -1 si la clé ne se trouve pas dans le tableau. Précisez la complexité de cette fonction.

Remarques :

— Pour les deux questions, vous pouvez supposer que le tableau ne contient que des valeurs uniques.

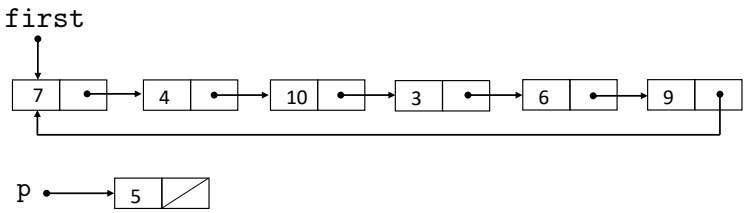
— Si besoin, vous pouvez réutiliser toutes les fonctions du cours sans les redéfinir.

Question 4 (liste liée)

Soit le type suivant servant à définir les nœuds d'une liste liée :

```
typedef struct Node_t {
  int value;
  struct Node_t *next;
} Node;
```

Supposons que la variable `first` contienne un pointeur vers un nœud de la liste liée (circulaire) représentée par cette figure :



et que `p` soit un nœud nouvellement créé par les instructions suivantes :

```
Node *p = malloc(sizeof(Node));
p->value = 5;
p->next = NULL;
```

Pour les 5 séquences d'instructions ci-dessous à gauche, supposées être exécutées indépendamment les unes des autres, identifiez la description correcte de leur effet sur la liste pointée par `first` dans la liste à droite.

1.
```
first->next = first->next->next;
```

2.
```
p->next = first->next;
first->next = p;
```

3.
```
p->next = first->next->next;
first->next = p;
```

4.
```
p->next->next = first->next->next;
first->next = p;
```

5.
```
Node *z = first;
while (z->next != first) z = z->next;
z->next = p;
p->next = first;
```

A. Pas de modification
B. Suppression du 7
C. Suppression du 4
D. Suppression du 10
E. Insertion de 5 après 7
F. Insertion de 5 après 4
G. Insertion de 5 après 9
H. Remplacement de 4 par 5
I. Remplacement de 7 par 5
J. `first` ne pointe plus vers une liste circulaire
K. Erreur à l'exécution

1 : _____ 2 : _____ 3 : _____ 4 : _____ 5 : _____

Question 5 (structure de données)

Répondez sur une feuille séparée à cette question.

Soit une structure de dictionnaire contenant des paires (clé, valeur) où les clés sont des chaînes de caractères et les valeurs sont de type `int` et supposées positives ou nulles (par exemple pour représenter le nombre d'occurrences de ces chaînes dans un document). Le fichier d'entête du dictionnaire est le suivant :

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

La fonction `dictSearch` renvoie la valeur associée à `key` dans `d` si elle existe, -1 sinon.
On souhaite ajouter à cette interface une fonction :

```
Dict *dictMerge(Dict *d1, Dict *d2);
```

prenant en argument deux dictionnaires et renvoyant un nouveau dictionnaire contenant l'ensemble des clés qui se trouvent dans l'un des deux dictionnaires. Lorsqu'une clé n'est présente que dans l'un des deux dictionnaires, elle doit conserver sa valeur dans le nouveau dictionnaire. Lorsqu'une clé est présente dans les deux dictionnaires, la valeur qui lui est associée est la somme des valeurs dans les deux dictionnaires.

1. Implémentez cette fonction dans le cas de l'implémentation par liste liée du dictionnaire. La structure de données utilisée pour le dictionnaire est la suivante :

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

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

   Vous pouvez utiliser les fonctions de l'interface et/ou travailler directement avec les champs de la structure de données. Les clés peuvent être partagées entre les dictionnaires mais les deux dictionnaires en argument ne doivent pas être modifiés.
2. En supposant que les deux ensembles ont la même taille n, donnez la complexité en temps en notation grand-O de votre implémentation. Justifiez votre réponse, en précisant à quoi correspond le pire cas.
3. Pensez-vous qu'il soit possible d'implémenter cette fonction plus efficacement (en notation grand-O) en utilisant une représentation par tableau trié et/ou par table de hachage ? Si oui, expliquez brièvement comment vous feriez (sans fournir de code).

Question 6 (débogage et maîtrise du c)

Répondez aux sous-questions 2 et 3 sur une feuille séparée.

Remarque : pour simplifier le code, la gestion d'un retour NULL de la fonction `malloc` *pourra être ignorée dans cette question.*

1. On souhaite implémenter un dictionnaire sur base d'un tableau trié *extensible*, c'est-à-dire dont la taille est doublée lorsqu'il n'y a plus de place pour une nouvelle clé (voir slide 327 du cours théorique). La page suivante reprend un extrait du fichier `dict.c` implémentant ce dictionnaire (le fichier `dict.h` est celui donné au slide 319 du cours). Il y a en fait trois erreurs dans la fonction `dictInsert`. Indiquez les le plus précisément possible directement sur le code et corrigez les.
2. Proposez une implémentation de la fonction de création de la structure compatible avec la fonction d'insertion.

   ```
   Dict *dictCreate() {
     // à compléter
     ...
   }
   ```

3. (Bonus) On aimerait utiliser cette implémentation d'un dictionnaire générique, utilisant des valeurs de type pointeur sur void, pour ré-implémenter l'application du comptage de mot (slide 322 du cours théorique) qui utilisait un dictionnaire à valeurs entières. Modifiez la séquence d'instructions ci-dessous, extraite de ce code, pour prendre en compte cette nouvelle implémentation du dictionnaire :

   ```
   int c = dictSearch(d, token);
   if (c == -1) {
     dictInsert(d, strdup(token), 1);
   } else {
     dictInsert(d, token, c+1);
   }
   ```

   La fonction `dictSearch` de l'implémentation à valeur de type pointeur sur void a le prototype suivant :

   ```
   void *dictSearch(Dict *d, char *key);
   ```

   Elle renvoie la valeur associée à la clé (un pointeur sur void) si la clé se trouve dans le dictionnaire, NULL si la clé ne s'y trouve pas.
   Suggestion : vous pouvez utiliser ce nouveau type pour stocker le nombre d'occurrences :

   ```
   typedef struct MyInt_t {
     int val;
   } MyInt;
   ```

```c
#include <stdlib.h>
#include <string.h>
#include "dict.h"

struct Dict_t {
  char **keys;     // tableau trié contenant les clés
  void **values;   // tableau contenant les valeurs correspondantes
  int size;        // taille du tableau
  int nKeys;       // nombre de clés stockées dans le tableau
};

void dictInsert(Dict *d, char *key, void *value) {
   if (d -> nKeys > 0) {
               // recherche binaire avec des clés à valeur 'char *'
      int pos = binary_search(key, d -> keys, d -> nKeys);
      if (pos != -1) {
         d -> values[pos] = value;
         return;
      }
   }
   if (d -> nKeys == d -> size) {
     d -> size = d -> size * 2;
     char **newKeys = malloc(d->size * sizeof(char));
     void **newValues = malloc(d->size * sizeof(void *));
     for (int i = 0; i < d -> nKeys; i++) {
       newKeys[i] = d -> keys[i];
       newValues[i] = d -> values[i];
     }
     free(d -> keys);
     free(d -> values);
     d -> keys = newKeys;
     d -> values = newValues;
   }
   int i = d -> nKeys;
   while (i >= 0 && strcmp(key, d -> keys[i-1]) < 0) {
     d -> keys[i] = d -> keys[i-1];
     d -> values[i] = d -> values[i-1];
     i--;
   }
   d -> keys[i] = key;
   d -> values[i] = value;
}
```