Complément d'informatique

Questions types d'examen

13 décembre 2018

Remarques :

— *Les questions ci-dessous sont à prendre comme des exemples de questions types pour l'examen écrit. La plupart des questions des TPs, sauf celles qui consistaient à réaliser des mesures de temps de calcul, constituent aussi des questions types potentielles de l'examen.*

— *Sauf mention explicite, toutes les complexités sont à décrire par rapport au temps d'exécution des opérations concernées dans le pire cas et en utilisant la notation grand-O. Le situation correspondant au pire cas doit à chaque fois être expliquée.*

Question 1 (récursivité)

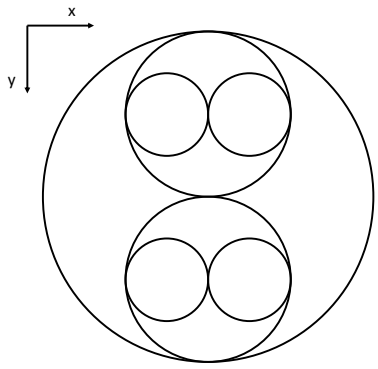
Soit le code suivant où l'appel `draw_circle(x,y,r)` trace un cercle de rayon `r` centré en (x, y).

```
void horiz(int n, double x, double y, double r) {
  if (n == 0) return;
  vert(n - 1, x + r/2, y, r/2);
  vert(n - 1, x - r/2, y, r/2);
  draw_circle(x, y, r);
}

void vert(int n, double x, double y, double r) {
  if (n == 0) return;
  horiz(n - 1, x, y + r/2, r/2);
  draw_circle(x, y, r);
  horiz(n - 1, x, y - r/2, r/2);
}

int main() {
  int n = ...;
  vert(n, .5, .5, .5);
}
```

Soit le graphique ci-dessous généré par ce code :



1. Quelle est la valeur de `n` dans la fonction main ?
2. Indiquer sur le graphique l'ordre d'affichage des cercles (1 pour le premier cercle à avoir été tracé et 7 pour le dernier).
3. Quelle est la complexité en temps de la fonction `vert` en fonction de son argument `n` ?

Question 2 (complexité)

Donnez les complexités en temps des fonctions `mystery1`, `mystery2`, et `mystery3` en fonction de la taille `n` du tableau donné en argument :

1.

```
int mystery1(int inArray[], int n) {
  int sum = 0;
  for (int i = 0; i < n; i++)
     for (int j = 0; j < i/2; j++)
        sum += inArray[j];
  for (int i = 0; i < n; i++)
     sum += inArray[i];
  return sum;
}
```

2.

```
int mystery2(int inArray[], int n) {
  int sum = 0;
  for (int i = 0; i < 10; i++)
     for (int j = 0; j < n; j++)
        sum += inArray[j];
  return sum;
}
```

3.

```
int mystery3_helper(int inArray[], int left, int right) {
  int sum = 0;
  int mid = ((right - left) / 2) + left;

  if (left == right) return 0;
  if (left + 1 == right) return 0;

  for (int i = left; i <= right; i++)
     sum += inArray[i];

  return mystery3_helper(inArray, left, mid)
       + mystery3_helper(inArray, mid, right)
       + sum;
}
int mystery3(int inArray[], int n) {
  return mystery3_helper(inArray, 0, n-1);
}
```

Question 3 (algorithmique)

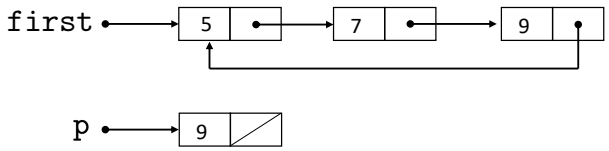
Soit la fonction suivante où `tab` est un tableau de `n` valeurs entières :

```
void algoX(int tab[], int n) {
  int i = 0;
  int j = 0;
  int maxcount = 0;
  int count = 0;
  while (i < n) {
    if (tab[i] == tab[j])
      count++;
    j++;
    if (j > n) {
      if (count > maxcount)
         maxcount = count;
      count = 0;
      i++;
      j = i;
    }
  }
  return maxcount;
}
```

1. Donnez la complexité en temps en notation grand-O de la fonction `algoX` en fonction de la taille du tableau `n`.
2. Expliquez brièvement ce que fait l'algorithme.
3. Ecrivez un algorithme de complexité strictement inférieure faisant la même chose que `algoX`. Vous pouvez ré-utiliser sans les redéfinir n'importe quel algorithme ou structure de données vue au cours.

Question 4 (liste liée)

Soit la liste suivante (circulaire) :



dont les noeuds sont du type suivant :

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

et soit les instructions suivantes :

A. `first = p;`
B. `first->next = p;`
C. `first->next->next = p;`
D. `first->next->next->next = p;`
E. `first->next->next->next->next = p;`
F. `p->next = first;`
G. `p->next = first->next;`
H. `p->next = first->next->next;`
I. `p->next = first->next->next->next;`
J. `p->next = first->next->next->next->next;`

Pour chaque opération ci-dessous, donnez la séquence d'instructions permettant de la réaliser en maintenant la liste circulaire (vous pouvez sélectionner chaque instruction zéro, une ou plus d'une fois).

1. Insérer le nœud pointé par `p` entre le premier et le second nœud.
2. Insérer le nœud pointé par `p` après le dernier nœud.
3. Insérer le nœud pointé par `p` entre le second et le troisième nœud.
4. Insérer le nœud pointé par `p` avant le premier nœud.

Question 5 (structure de données)

Soit la structure d'ensemble vue au cours dont l'interface est donnée au slide 319 (partie 6). On veut ajouter à cette interface une fonction :

```
Set *setDifference(Set *s1, Set *s2);
```

renvoyant un nouvel ensemble contenant les éléments qui se trouvent dans `s1` mais pas dans `s2`.

1. Implémentez cette fonction dans le cas de l'implémentation par liste liée de l'ensemble et des clés à valeurs entières. La structure de données utilisée est la suivante :

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

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

   Vous pouvez utiliser les fonctions de l'interface et/ou travailler directement sur la structure de liste liée.
2. En supposant que les deux ensembles ont la même taille n, donnez la complexité en temps en notation grand-O de cette implémentation.
3. Quelle serait la complexité de cette fonction pour les représentations par tableau trié et par table de hachage de l'ensemble ?

Questions 6 (maîtrise du c)

Soit le nouveau type suivant contenant les informations relatives à une ville :

```
typedef struct Town_t {
  char *name;
  double x;
  double y;
} Town;
```

1. Répérez les trois erreurs dans les fonctions suivantes de création et de libération mémoire associées.

```
Town *createTown(char *name, double x, double y) {
  Town *town = malloc(sizeof(Town *));

  if (town == NULL) exit(-1);

  int lengthName=0;
  while (name[lengthName] != '\0')
    lengthName++;
  town -> name = malloc(lengthName*sizeof(char));

  if (town -> name == NULL) {
    free(town);
    exit(-1);
  }

  int i;
  for (i = 0; i<lengthName; i++)
    town -> name[i] = name[i];
  town->name[i] = '\0';

  town -> x = x;
  town -> y = y;

  return town;
}

void freeTown(Town *t) {
  free(t);
  free(t->name);
}
```

2. On souhaite utiliser la fonction de tri générique vue au cours (Partie 3, slide 145) pour trier les villes par rapport à leur distance euclidienne à une ville source. Complétez la fonction `compare_town` dans le code suivant dans ce but.

```
#include <stdio.h>
#include <stdlib.h>
#include "sort.h"
#include "town.h"

int compare_town(void *array, int i, int j);
void swap_town(void *array, int i, int j);

void swap_town(void *array, int i, int j) {
  Town *temp = ((Town **)array)[i];
  ((Town **)array)[i] = ((Town **)array)[j];
  ((Town **)array)[j] = temp;
}

Town *source;

int compare_town(void *array, int i, int j) {
  // A compléter
  ...
}

int main() {

  Town *liege = createTown("Liège",85.7,22.3);
  Town *anvers = createTown("Anvers",3.4,-41.8);
  Town *bruxelles = createTown("Bruxelles",0.0,0.0);
  Town *arlon = createTown("Arlon",102.5,124.7);
  Town *namur = createTown("Namur",34.9,43.1);

  Town *tab[5];

  tab[0] = anvers;
  tab[1] = bruxelles;
  tab[2] = arlon;
  tab[3] = namur;

  source = liege;

  sort(tab, 4, compare_town, swap_town);

  for (int i = 0; i < 4; i++)
    printf("%s \n ",tab[i]->name);
```

```
  freeTown(liege);
  freeTown(anvers);
  freeTown(bruxelles);
  freeTown(arlon);
  freeTown(namur);

  exit(0);
}
```