# 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)

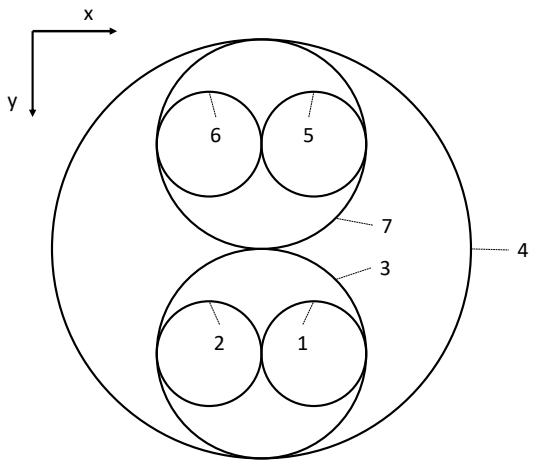
Soit le code suivant où l'appel `draw_circle(x,y,r)` trace un cercle de rayon `r` centré en $(x, y)$.

```
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` ?

— Réponse —

1. `n` = 3.

2. La trace des appels récursifs est la suivante :

```
vert(3, .5, .5, .5)
 |horiz(2, .5, .75, .25)
 |  |vert(1, .625, .75, .125)
 |  |  |horiz(0,...)
 |  |  |draw_circle(.625, .75, .125)
 |  |  |horiz(0,...)
 |  |vert(1, .375, .75, .125)
 |  |  |horiz(0,...)
 |  |  |draw_circle(.375, .75, .125)
 |  |  |horiz(0,...)
 |  |draw_circle(.5, .75, .25)
 |draw_circle(.5, .5, .5)
 |horiz(2, .5, .25, .25)
 |  |vert(1, .625, .25, .125)
 |  |  |horiz(0,...)
 |  |  |draw_circle(.625, .25, .125)
 |  |  |horiz(0,...)
 |  |vert(1, .375, .25, .125)
 |  |  |horiz(0,...)
 |  |  |draw_circle(.375, .25, .125)
 |  |  |horiz(0,...)
 |  |draw_circle(.5, .25, .25)
```

ce qui donne l'ordre de tracé des cercles représenté sur la figure ci-dessus.

3. Soit $T(n)$ les temps de calcul en fonction de $n$. On a $T(n) = 2T(n-1) + c$, avec $T(0) = d$, où $c$ et $d$ sont des constantes ne dépendant pas de $n$. En développant l'équation récursive :

$$\begin{aligned} T(n) &= 2T(n-1) + c && (1)\\ &= 2^2T(n-2) + 2c + c && (2)\\ &= 2^3T(n-3) + 2^2c + 2c + c && (3)\\ &= \ldots && (4)\\ &= 2^kT(n-k) + c\sum_{i=0}^{k-1} 2^i && (5)\\ &= 2^kT(n-k) + c(2^k - 1) && (6) \end{aligned}$$

En prenant $k = n$, on obtient :

$$\begin{aligned} T(n) &= 2^nT(0) + c(2^n - 1) && (7)\\ &= (c+d)2^n - c && (8) \end{aligned}$$

et donc $T(n) \in O(2^n)$. On aurait pu également observer que la structure des appels récursifs, et donc la complexité, est identique à celle de la fonction `hanoi` vue au cours.

## 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);
}
```

— Réponse —

1. $O(n^2)$
2. $O(n)$
3. $O(n \log n)$ (par analogie au mergesort).

## Question 3 (algorithmique)

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

```
int 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.

— Réponse —

1. $O(n^2)$
2. La fonction renvoie le nombre d'occurrences de l'entier qui apparaît le plus fréquemment dans le tableau `tab`.
3. La fonction suivante renvoie le même résultat que `algoX`. L'idée est de d'abord trier le tableau et d'ensuite compter en une seule passe sur le tableau le nombre d'occurrences de chaque élément unique du tableau.

   ```
   int algoXnew(int tab[], int n) {
     int *newtab = malloc(n*sizeof(int));
     for (int i = 0; i < n; i ++)
       newtab[i] = tab[i];
     mergesort(newtab, n);
     int maxcount = 0;
     int i = 0;
   ```

```
  while (i < n) {
    int count = 1;
    int j = i+1;
    while (j < n && tab[j] == tab[j-1]) {
      count++;
      j++;
    }
    if (count > maxcount)
      maxcount = count;
    i = j;
  }
  return maxcount;
}
```

Complexité : $O(n \log n)$ ($O(n \log n)$ pour le tri et $O(n)$ pour la boucle `while` qui suit).

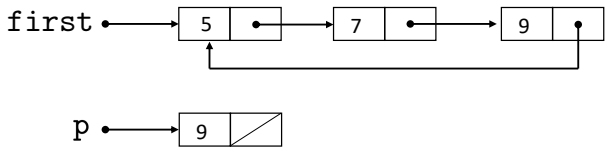
Une autre solution serait d'utiliser une table de hachage :

```
int algoXnew(int tab[], int n) {
  Dict *d = dictCreate(n*2);
  int maxcount = 0;
  for (int i=0; i < n; i++) {
    int c = dictSearch(d, tab[i]);
    if (c == -1) {
      dictInsert(d, tab[i], 1);
    } else {
      dictInsert(d, tab[i], c+1);
      if (c+1 > maxcount) {
        maxcount = c+1;
      }
    }
  }
  dictFree(d);
  return maxcount;
}
```

En pratique, si la fonction de hachage est bien choisie, l'insertion sera $O(1)$ et donc l'algorithme sera $O(n)$. Cependant, dans le pire cas (par exemple si toutes les valeurs du tableau sont différentes et sont envoyées dans la même case de la table de hachage), la complexité sera $O(n^2)$ et donc cette solution n'est pas meilleure que la version de l'énoncé.

## 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.

— Réponse —

1. GB
2. FD
3. HC
4. FDA ou bien DFA.

## 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 ?

— Réponse —

1. ```
   Set *setDifference(Set *s1, Set *s2) {
     Set *newset = setCreate();
     Node *p = s1 -> first;
     while (p!=NULL) {
       if (!setContains(s2, p -> key)) {
         setInsert(newset, p -> key);
       }
       p = p -> next;
     }
     return newset;
   }
   ```

2. $O(n^2)$.
   — Le pire cas correspond par exemple au cas où aucun élément de `s1` n'apparaît dans `s2`. Il faut donc parcourir pour chaque élément de `s1` l'intégralité de `s2`.
3. Vecteur trié : $O(n \log n)$
   ```

— `setContains` est $O(\log n)$ et `setInsert` est $O(n)$ dans le pire cas ($O(\log n)$ pour rechercher la position du nouvel élément et puis $O(n)$ pour l'insertion au début du vecteur à cause du décalage). La boucle est donc $O(n^2)$. Ce n'est cependant pas la borne supérieure la plus serrée. En effet, comme les éléments de `s1` sont parcourus dans l'ordre, ils seront également insérés en ordre dans le nouvel ensemble. Ainsi le nouvel élément sera toujours ajouté à la fin du vecteur et aucun décalage ne devra être fait. Dans le contexte de cet algorithme, `setInsert` sera donc $O(\log n)$ et donc `setDifference` sera $O(n \log n)$. ($O(n^2)$ aurait été accepté comme solution également.)

Table de hachage : $O(n^2)$.

— Dans le pire cas, toutes les valeurs de `s2` sont envoyées dans la même case de la table de hachage et donc `setContains` est $O(n)$. Si l'intersection de `s1` et `s2` est en plus vide, la boucle sera $O(n^2)$.

## 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.

```
1  Town *createTown(char *name, double x, double y) {
2    Town *town = malloc(sizeof(Town *));
3
4    if (town == NULL) exit(-1);
5
6    int lengthName=0;
7    while (name[lengthName] != '\0')
8      lengthName++;
9    town -> name = malloc(lengthName*sizeof(char));
10
11   if (town -> name == NULL) {
12     free(town);
13     exit(-1);
14   }
15
16   int i;
17   for (i = 0; i<lengthName; i++)
18     town -> name[i] = name[i];
19   town->name[i] = '\0';
20
21   town -> x = x;
22   town -> y = y;
23
24   return town;
25 }
26
27 void freeTown(Town *t) {
28   free(t);
29   free(t->name);
30 }
```

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);
}
```

— Réponse —

1. Les erreurs sont les suivantes :
   (a) Ligne 2 : `sizeof(Town *)` devrait être `sizeof(Town)`. On souhaite allouer une structure de type `Town` et pas un pointeur vers un structure de type `Town`.
   (b) Ligne 19 : cette ligne générera une erreur à l'exécution : la taille du tableau de caractères alloué à la ligne 9 n'inclut pas l'espace mémoire pour le caractère zéro de fin de chaîne.
   (c) Ligne 28-29 : ces deux instructions sont inversées. Il faut d'abord libérer `t->name` avant de libérer `t`.
2. 
```
int compare_town(void *array, int i, int j) {
  Town *ti = ((Town **)array)[i];
  Town *tj = ((Town **)array)[j];
  double dti = (ti->x - source->x)*(ti->x - source->x)
             + (ti->y - source->y)*(ti->y - source->y);
  double dtj = (tj->x - source->x)*(tj->x - source->x)
             + (tj->y - source->y)*(tj->y - source->y);
  return (dti <= dtj);
}
```