Cours d'introduction à l'informatique
Examen de seconde session 2024
Énoncés et solutions

Énoncés

1. (a) Écrire une fonction prenant en arguments un tableau t d'entiers signés supposés triés par ordre croissant, ainsi que la taille n de ce tableau, et retournant le nombre de valeurs différentes parmi les éléments de t. Par exemple, si t contient $[-1, -1, 4, 5, 5]$, alors la fonction doit retourner 3. Si $n = 0$, la fonction doit retourner 0.
   (b) Par la méthode des invariants, démontrer que la valeur retournée par cette fonction est correcte.
   (c) Calculer la complexité en temps de cette fonction.

2. Le *triangle d'Euler* est une construction mathématique constituée d'une séquence de lignes de longueur croissante : pour $i = 1, 2, \ldots$, la ligne d'indice i contient i nombres entiers $A_{i,0}$, $A_{i,1}$, …, $A_{i,i-1}$ définis de la façon suivante :
   — Le premier et le dernier élément de chaque ligne sont égaux à 1 :
   $$\forall i > 0 : \; A_{i,0} = A_{i,i-1} = 1.$$
   — Les autres éléments sont déterminés par l'équation
   $$\forall 0 < j < i - 1 : \; A_{i,j} = (i - j)A_{i-1,j-1} + (j + 1)A_{i-1,j}.$$
   (a) Écrire un programme qui affiche à l'écran les N premières lignes du triangle d'Euler, où $N \geq 1$ est une constante définie au début de ce programme.

   Par exemple, pour $N = 6$, le programme doit produire l'affichage suivant :

```
1
1 1
1 4 1
1 11 11 1
1 26 66 26 1
1 57 302 302 57 1
```

   (b) Calculer la complexité en temps et en espace du programme obtenu au point (a).

3. (a) Décrire le plus simplement possible ce que calcule la fonction C suivante :

```
unsigned f(unsigned i, unsigned j)
{
    if (i < j)
      return f(j, i);
    if (j == 0)
      return 0;
    return i + f(i, j-1);
}
```

   (b) Quelle est la complexité en espace de cette fonction ?

   (c) Écrire une fonction C réalisant exactement la même opération que la fonction du point (a), mais sans effectuer d'appel récursif.

4. Une application de gestion de notes d'étudiants représente ces étudiants comme les éléments d'une liste simplement liée. Chacun de ces éléments est composé du matricule d'un étudiant, formé par 8 caractères, et d'un pointeur vers l'élément suivant dans la liste, ce pointeur étant vide pour le dernier élément. La liste liée retient uniquement un pointeur vers son premier élément (qui est vide si cet élément n'existe pas), ainsi que le nombre d'éléments qu'elle contient.

   (a) Écrire un fragment de code C définissant deux types structurés : le premier pour un élément de liste liée représentant un étudiant, et le second pour une liste simplement liée représentant une séquence d'étudiants.

   (b) Écrire une fonction prenant en arguments un pointeur vers la représentation d'une liste liée ℓ d'étudiants, et un tableau de caractères contenant le matricule m d'un étudiant supplémentaire. Si le matricule m n'est pas déjà présent dans ℓ, alors la fonction doit ajouter l'étudiant correspondant à la fin de cette liste et retourner 0. Sinon, la fonction doit laisser ℓ inchangée et retourner -1.

   Notes : Les types des éléments de la liste liée et de la liste liée elle-même doivent correspondre à votre réponse au point (a). Vous pouvez programmer des fonctions ou des types de données supplémentaires si votre solution le nécessite.

   (c) Écrire une fonction prenant en argument un pointeur vers la représentation d'une liste liée, compatible avec votre réponse au point (a), et libérant l'ensemble de la mémoire allouée à cette liste et à ses éléments.

Exemples de solutions

1. (a) Il suffit de parcourir les paires d'éléments consécutifs de t, et d'incrémenter un compteur chaque fois que l'on détecte deux éléments différents.

```
unsigned nb_differents(int t[], unsigned n)
{
  unsigned i, c;

  if (!n)
    return 0;

   for (i = 0, c = 1; i + 1 < n; i++)
     if (t[i] != t[i + 1])
       c++;

  return c;
}
```

   (b) On souhaite établir la validité du triplet suivant :

$$\{\texttt{t}[0 : \texttt{n} - 1] \text{ trié}\}$$

```
if (!n)
  return 0;

for (i = 0, c = 1; i + 1 < n; i++)
  if (t[i] != t[i + 1])
    c++;
return c;
```

$$\{\text{Valeur de retour} = \text{nombre d'éléments différents dans } \texttt{t}[0 : \texttt{n} - 1]\},$$

   où la notation $\texttt{t}[a : b]$ dénote le sous-tableau de `t` formé par les éléments dont l'index est compris entre a et b (inclus).

   On distingue deux cas. Si $\texttt{n} = 0$, la valeur retournée 0 est correcte. Il reste donc à considérer le cas $\texttt{n} > 0$, correspondant au triplet suivant :

$$\{\texttt{n} > 0,\ \texttt{t}[0 : \texttt{n} - 1] \text{ trié}\}$$

```
for (i = 0, c = 1; i + 1 < n; i++)
  if (t[i] != t[i + 1])
    c++;
```

$$\{\texttt{c} = \text{nombre d'éléments différents dans } \texttt{t}[0 : \texttt{n} - 1]\}$$

En décomposant la boucle `for`, ce triplet devient :

$$\{\texttt{n} > 0,\ \texttt{i} = 0,\ \texttt{c} = 1,\ \texttt{t}[0 : \texttt{n} - 1] \text{ trié}\}$$

```
while (i + 1 < n)
  {
    if (t[i] != t[i + 1])
      c++;
    i++;
  }
```

$$\{\texttt{c} = \text{nombre d'éléments différents dans } \texttt{t}[0 : \texttt{n} - 1]\}$$

Pour obtenir un invariant de boucle I, on cherche à caractériser le travail effectué par la boucle jusqu'à l'itération courante. Un invariant possible est

$$I : 0 \leq \texttt{i} \leq \texttt{n} - 1 \text{ et } \texttt{t}[0 : \texttt{n} - 1] \text{ trié et}$$
$$\texttt{c} = \text{nombre d'éléments différents dans } \texttt{t}[0 : \texttt{i}].$$

Montrons maintenant que cet invariant est valide.

— Initialement, on a $\texttt{i} = 0$ et $\texttt{c} = 1$, donc l'invariant I est satisfait, car le tableau `t` limité à son premier élément contient exactement un élément.

— Pour chaque itération de la boucle, on a le triplet

$$\{I,\ \texttt{i} + 1 < \texttt{n}\}$$

```
if (t[i] != t[i + 1])
  c++;
i++;
```

$$\{I\}$$

Montrons que ce triplet est valide, en notant respectivement `x` et $\texttt{x}'$ la valeur d'une variable `x` avant et après l'exécution du code. Premièrement, on a $\texttt{i}' = \texttt{i} + 1$, qui combiné avec la précondition $\texttt{i} + 1 < \texttt{n}$ et $\texttt{n}' = \texttt{n}$ entraîne $\texttt{i}' \leq \texttt{n}' - 1$. Ensuite, il y a deux cas à considérer :

— Si $\mathtt{t}[\mathtt{i}] = \mathtt{t}[\mathtt{i}+1]$, alors les tableaux $\mathtt{t}[0 : \mathtt{i}]$ et $\mathtt{t}[0 : \mathtt{i}']$ contiennent le même nombre d'éléments différents, et l'on a $\mathtt{c}' = \mathtt{c}$. La post-condition est donc satisfaite.

— Si $\mathtt{t}[\mathtt{i}] \neq \mathtt{t}[\mathtt{i}+1]$, alors la précondition stipulant que $\mathtt{t}$ est trié conduit à ce que $\mathtt{t}[0 : \mathtt{i}']$ contienne un élément différent de plus que $\mathtt{t}[0 : \mathtt{i}]$. On a bien dans ce cas $\mathtt{c}' = \mathtt{c}+1$, et la postcondition est satisfaite.

— En sortie de boucle, on a $\{I, \mathtt{i}+1 \geq \mathtt{n}\}$, qui implique $\mathtt{i} = \mathtt{n}-1$. Il découle alors de l'invariant que $\mathtt{c}$ contient le nombre d'éléments différents contenus dans $\mathtt{t}[0 : \mathtt{n}-1]$, ce qui correspond bien à la postcondition.

(c) Le nombre d'itérations de la boucle est $O(\mathtt{n})$, et chacune d'entre elles effectue un nombre borné d'opérations. La complexité en temps vaut donc $O(\mathtt{n})$.

2. (a) On peut résoudre le problème en gérant un seul tableau qui contiendra les lignes successives du triangle d'Euler, mais il est alors nécessaire de garder dans des variables supplémentaires des copies des éléments de la ligne $i-1$ qui sont écrasés lors du calcul de la ligne i.

```
#include <stdio.h>
#define N 6

int main()
{
  unsigned a[N], i, j, aj, ajm1;

  for (i = 1; i <= N; i++)
    {
      a[0] = 1;
      a[i - 1] = 1;

      for (j = 1, ajm1 = 1; j < i-1; j++)
        {
          aj = a[j];
          a[j] = (i - j) * ajm1 + (j + 1) * aj;
          ajm1 = aj;
        }
```

```
      for (j = 0; j < i; j++)
        printf(" %u", a[j]);
      printf("\n");
    }

  return 0;
}
```

(b) Le paramètre du problème est `N`. Le nombre d'itérations de chacune des deux boucles imbriquées est $O(\mathtt{N})$, ce qui conduit à une complexité en temps égale à $O(\mathtt{N}^2)$. Étant donné que l'on ne gère qu'un tableau de taille `N`, en plus de variables dont la taille est bornée indépendamment de `N`, et qu'il n'y a pas de récursion, la complexité en espace vaut $O(\mathtt{N})$.

3. (a) Cette fonction retourne le produit de `i` et `j`.

(b) Si $\mathtt{i} \geq \mathtt{j}$, alors la profondeur de récursion est égale à $\mathtt{j} + 1$. Si $\mathtt{i} < \mathtt{j}$, elle est égale à $\mathtt{i} + 2$. La complexité en espace vaut donc $O(\min(\mathtt{i}, \mathtt{j}))$.

(c)
```
unsigned f(unsigned i, unsigned j)
{
  return i * j;
}
```

4. (a)
```
struct etudiant
{
  char matricule[8];
  struct etudiant *suivant;
};

struct liste_etudiants
{
  struct etudiant *premier;
  unsigned nb_elements;
};
```

(b) On définit une fonction auxiliaire capable de comparer deux matricules. L'énoncé ne précisant pas le comportement attendu de la fonction dans le cas d'une insuffisance mémoire, on choisit de retourner -1 en laissant la liste inchangée, comme dans le cas d'un matricule déjà présent. Enfin, on retient dans une variable auxiliaire `pe` l'endroit où est stocké le pointeur

vers le dernier élément visité de la liste, afin de pouvoir facilement le modifier pour ajouter un élément supplémentaire en fin de liste.

```
#include <stdlib.h>
#include <string.h>

#define L_MATRICULE  8

int matricules_egaux(char m1[], char m2[])
{
  unsigned i;

  for (i = 0; i < L_MATRICULE; i++)
    if (m1[i] != m2[i])
      return 0;

  return 1;
}

int ajouter_etudiant(struct liste_etudiants *l, char m[])
{
  struct etudiant *e, **pe;

  for (e = l -> premier, pe = &l -> premier; e;
       pe = &e -> suivant, e = *pe)
    if (matricules_egaux(m, e -> matricule))
      return -1;

  e = malloc(sizeof(struct etudiant));
  if (!e)
    return -1;

  memcpy(e -> matricule, m, L_MATRICULE);
  e -> suivant = NULL;
  *pe = e;
  l -> nb_elements++;

  return 0;
}
```

(c) Pour cette question, on suppose que la structure contenant la liste et celles contenant ses éléments ont été allouées individuellement.

```
#include <stdlib.h>

void liberer_liste(struct liste_etudiants *l)
{
  struct etudiant *e, *e_suivant;

  for (e = l -> premier; e; e = e_suivant)
    {
      e_suivant = e -> suivant;
      free(e);
    }

  free(l);
}
```