Cours d'introduction à l'informatique
Examen de janvier 2025
Énoncés et solutions

Énoncés

1. (a) Un tableau est dit *monotone* si ses éléments sont triés par ordre croissant ou décroissant. Par exemple, les tableaux $[2, 0, 0, -1]$, $[42]$ et $[10, 10, 20, 30]$ sont monotones, mais les tableaux $[23, 1, 2025]$ et $[-1, 1, 1, -1]$ ne le sont pas.

   Écrire une fonction qui prend en arguments un tableau d'entiers `t` et sa taille `n`, et qui retourne 1 si `t` est monotone et 0 sinon.

   (b) Par la méthode des invariants, démontrer que la valeur calculée par cette fonction est correcte.

2. (a) Écrire une procédure prenant en arguments un tableau d'entiers non signés `t`, sa taille `n`, un tableau `u`, et la taille `m` de ce dernier. On suppose que les éléments de `t` sont tous différents et que l'on a $\texttt{m} \leq \texttt{n}$. La procédure demandée doit écrire dans `u` les `m` plus petits éléments de `t`, dans un ordre quelconque. Le contenu de `t` doit rester inchangé.

   Par exemple, si `t` contient $[1, 5, 3, 0, 4, 7]$ et $\texttt{m} = 3$, alors la procédure doit écrire dans `u` les valeurs 0, 1 et 3.

   Note : On ne demande pas la solution la plus efficace possible ; un algorithme polynomial suffit.

   (b) Calculer la complexité en temps de la procédure obtenue au point (a).

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

```
unsigned f(unsigned a, unsigned b)
{
  if (a < b || b == 0)
    return 0;

  return 1 + f(a - b, b);
}
```

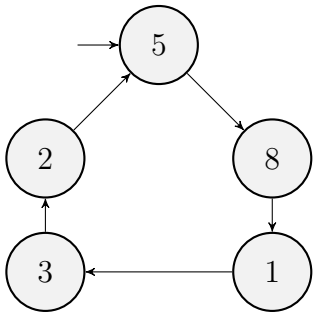
(b) Quelle est la complexité en espace de cette fonction ? (Justifier votre réponse.)

(c) Écrire une fonction réalisant exactement la même opération, mais sans effectuer d'appel récursif ni d'itération de boucle.

4. Un *graphe cyclique* de taille $n > 0$ est composé d'un ensemble de n *nœuds*, chacun d'entre eux étant caractérisé par un *identifiant* entier qui lui est propre et par un pointeur vers un nœud appelé son *successeur*. Un des nœuds du graphe est distingué comme étant *initial*.

   Pour être valide, le graphe doit former un unique cycle. Cela signifie que si on le parcourt en passant répétitivement d'un nœud à son successeur, en commençant à partir d'un nœud quelconque, on finira toujours par visiter tous les nœuds.

   Par exemple, la figure suivante montre un graphe cyclique de taille 5, dont les identifiants des nœuds forment l'ensemble $\{1, 2, 3, 5, 8\}$. Le successeur de chaque nœud est représenté par une flèche, et la flèche sans origine indique que le nœud initial est celui d'identifiant 5.



   (a) Écrire des fragments de code définissant deux types structurés permettant de représenter un nœud d'un graphe cyclique, et un graphe cyclique. On demande que chaque champ de ces types structurés corresponde soit à l'identifiant d'un nœud, soit à un pointeur vers un nœud.

   (b) Écrire une fonction prenant en arguments un tableau d'identifiants et la longueur de ce tableau, et retournant un pointeur vers une représentation nouvellement allouée d'un graphe cyclique dont les nœuds successifs correspondent aux éléments du tableau. Le premier élément du tableau définit le nœud initial. Par exemple, si la fonction reçoit en entrée le tableau $[\,5, 8, 1, 3, 2\,]$, alors elle doit produire le graphe cyclique de la figure précédente. Le type de la représentation générée doit bien sûr être compatible avec votre réponse au point (a).

(c) Écrire une procédure acceptant en argument un pointeur vers une représentation d'un graphe cyclique, chargée d'afficher dans l'ordre l'identifiant de tous les nœuds visités lors d'un parcours de ce graphe. Par exemple, pour le graphe de la figure précédente, elle doit afficher "`5 8 1 3 2`". On demande également que cette procédure libère tout l'espace mémoire occupé par la représentation du graphe.

Note : Vous pouvez programmer des fonctions ou des types de données supplémentaires si votre solution le nécessite.

Exemples de solutions

1. (a) On peut résoudre ce problème en inspectant toutes les paires d'éléments consécutifs dans le tableau, tout en gérant une variable `ordre` qui indique dans quel ordre ces éléments sont triés. On peut par exemple considérer qu'`ordre` $= 1$ indique un tri par ordre croissant, `ordre` $= -1$ un tri par ordre décroissant, et `ordre` $= 0$ une situation où l'ordre n'est pas encore déterminé. On obtient le code suivant :

```
int est_monotone(int t[], unsigned n)
{
  int ordre;
  unsigned i;

  for (i = 0, ordre = 0; i + 1 < n; i++)
    {
      if ((ordre == 1 && t[i + 1] < t[i]) ||
          (ordre == -1 && t[i + 1] > t[i]))
        return 0;

      if (t[i + 1] > t[i])
        ordre = 1;
      else
        if (t[i + 1] < t[i])
          ordre = -1;
    }

  return 1;
}
```

(b) Pour écrire un triplet décrivant que le code obtenu en (a) retourne un résultat correct, on peut introduire une variable fictive `r` représentant la valeur de retour de la fonction. On souhaite alors établir que le triplet suivant est valide :

$$\{\mathtt{n} \geq 0, \mathtt{r} = 1\}$$

```
for (i = 0, ordre = 0; i + 1 < n; i++)
  {
    if ((ordre == 1 && t[i + 1] < t[i]) ||
        (ordre == -1 && t[i + 1] > t[i]))
      {
        r = 0;
        break;
      }

    if (t[i + 1] > t[i])
      ordre = 1;
    else
      if (t[i + 1] < t[i])
        ordre = -1;
  }
```

$\{\mathtt{r} \neq 0$ ssi le tableau $\mathtt{t}[0 : \mathtt{n} - 1]$ est monotone$\}$

On peut se débarrasser de l'instruction de rupture de séquence `break` en ajoutant au gardien de la boucle une condition sur la variable `r`. On en profite pour également décomposer l'instruction `for`. On obtient le triplet équivalent suivant :

$$\{\mathtt{n} \geq 0, \mathtt{i} = 0, \mathtt{ordre} = 0, \mathtt{r} = 1\}$$

```
while (i + 1 < n && r != 0)
  {
    if ((ordre == 1 && t[i + 1] < t[i]) ||
        (ordre == -1 && t[i + 1] > t[i]))
      r = 0;
    else
      {
        if (t[i + 1] > t[i])
          ordre = 1;
        else
```

```
        if (t[i + 1] < t[i])
          ordre = -1;
      i++;
    }
}
```

$\{\texttt{r} \neq 0$ ssi le tableau $\texttt{t}[0 : \texttt{n} - 1]$ est monotone$\}$

Pour obtenir un invariant de boucle I, on cherche à caractériser les opérations effectuées par la boucle jusqu'à l'itération courante. Dans le cas particulier où $\texttt{n} = 0$, le tableau est par définition monotone. Notre spécification des valeurs que prennent les variables `ordre` et `r` mène au candidat invariant suivant :

I : ($\texttt{n} = 0$ et $\texttt{i} = 0$ et $\texttt{r} \neq 0$) ou
($0 \leq \texttt{i} \leq \texttt{n} - 1$ et
(($\texttt{r} \neq 0$ et $\texttt{ordre} = 1$ et $\texttt{t}[0 : \texttt{i}]$ est croissant) ou
($\texttt{r} \neq 0$ et $\texttt{ordre} = -1$ et $\texttt{t}[0 : \texttt{i}]$ est décroissant) ou
($\texttt{r} \neq 0$ et $\texttt{ordre} = 0$ et $\texttt{t}[0 : \texttt{i}]$ est constant) ou
($\texttt{r} = 0$ et $\texttt{t}[0 : \texttt{n} - 1]$ n'est pas monotone)))

Dans cette expression, on utilise la convention habituelle de dénoter par $\texttt{t}[a : b]$ le sous-tableau de `t` formé par les éléments dont l'index est compris entre a et b (inclus).

Montrons maintenant que cet invariant I est valide.

— Initialement, on a $\texttt{n} \geq 0$, $\texttt{i} = 0$ et $\texttt{r} \neq 0$, donc l'invariant est satisfait, car un tableau croissant, décroissant ou constant est par définition monotone.

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

$\{I,\ \texttt{i} + 1 < \texttt{n},\ \texttt{r} \neq 0\}$

```
if ((ordre == 1 && t[i + 1] < t[i]) ||
    (ordre == -1 && t[i + 1] > t[i]))
  r = 0;
else
  {
    if (t[i + 1] > t[i])
      ordre = 1;
```

```
    else
      if (t[i + 1] < t[i])
        ordre = -1;
    i++;
  }
```

$\{I\}$

Notons comme on le fait habituellement `x` et $\mathtt{x}'$ les valeurs d'une variable `x` respectivement avant et après l'exécution du fragment de code.

La précondition ne permet pas d'avoir $\mathtt{n} = 0$, car $0 \leq \mathtt{i} < \mathtt{n} - 1$ n'est alors pas satisfait. On a donc $\mathtt{n} > 0$ et la précondition $\mathtt{i} + 1 < \mathtt{n}$ implique que l'on a bien $\mathtt{i}' + 1 \leq \mathtt{n}'$. En effet, on a $\mathtt{n}' = \mathtt{n}$, et soit $\mathtt{i}' = \mathtt{i}$ soit $\mathtt{i}' = \mathtt{i} + 1$. De plus la précondition $\mathtt{r} \neq 0$ combinée à l'invariant I implique qu'initialement, le tableau $\mathtt{t}[0 : \mathtt{i}]$ est croissant si $\mathtt{ordre} = 1$, décroissant si $\mathtt{ordre} = -1$, ou constant si $\mathtt{ordre} = 0$. Il y a plusieurs cas à considérer :

- Si $\mathtt{ordre} \in \{-1, 1\}$ et l'ordre entre $\mathtt{t}[\mathtt{i}]$ et $\mathtt{t}[\mathtt{i} + 1]$ est contraire à l'ordre entre les éléments de $\mathtt{t}[0 : \mathtt{i}]$, alors le tableau $\mathtt{t}[0 : \mathtt{i} + 1]$ n'est pas monotone, ce qui signifie que le tableau $\mathtt{t}[0 : \mathtt{n}']$ ne l'est pas non plus étant donné qu'il contient un sous-tableau non monotone. Le code produit alors $\mathtt{r}' = 0$ et la postcondition est satisfaite.
- Si $\mathtt{ordre} \in \{-1, 1\}$ et l'ordre entre $\mathtt{t}[\mathtt{i}]$ et $\mathtt{t}[\mathtt{i} + 1]$ est conforme à l'ordre entre les éléments de $\mathtt{t}[0 : \mathtt{i}]$, alors le seul effet du code est d'incrémenter `i`, et la postcondition est satisfaite.
- Si $\mathtt{ordre} = 0$ et $\mathtt{t}[\mathtt{i} + 1] > \mathtt{t}[\mathtt{i}]$, alors $\mathtt{i}' = \mathtt{i} + 1$ et $\mathtt{t}[0 : \mathtt{i}']$ est croissant. On a alors $\mathtt{ordre}' = 1$, ce qui satisfait la postcondition.
- Si $\mathtt{ordre} = 0$ et $\mathtt{t}[\mathtt{i} + 1] < \mathtt{t}[\mathtt{i}]$, alors $\mathtt{i}' = \mathtt{i} + 1$ et $\mathtt{t}[0 : \mathtt{i}']$ est décroissant. On a alors $\mathtt{ordre} = -1$, ce qui satisfait la postcondition.
- Si $\mathtt{ordre} = 0$ et $\mathtt{t}[\mathtt{i} + 1] = \mathtt{t}[\mathtt{i}]$, alors $\mathtt{i}' = \mathtt{i} + 1$ et $\mathtt{t}[0 : \mathtt{i}']$ est constant. On a alors $\mathtt{ordre}' = \mathtt{ordre} = 0$, ce qui satisfait la postcondition.

— En fin de boucle, on a $\{I, (\mathtt{i} + 1 \geq \mathtt{n}$ ou $\mathtt{r} = 0)\}$.

- Si $\mathtt{r} = 0$, alors l'invariant implique que le tableau $\mathtt{t}[0 : \mathtt{n} - 1]$ n'est pas monotone. La postcondition est donc satisfaite.

— Si $\mathtt{r} \neq 0$, alors on a $\mathtt{i} \leq \mathtt{n}-1$ et $\mathtt{i}+1 \geq \mathtt{n}$ qui entraînent $\mathtt{i} = \mathtt{n}-1$. L'invariant implique alors que le tableau $\mathtt{t}[0 : \mathtt{n}-1]$ est croissant, décroissant ou constant, donc monotone. La postcondition est alors satisfaite.

2. (a) On peut écrire une boucle qui effectue `m` itérations, chacune étant chargée de trouver dans le tableau `t` un nouvel élément à écrire dans `u`. On s'aide d'une variable `seuil` qui contient à chaque itération la valeur minimum de l'élément recherché, et d'une variable `min` qui retient la plus petite valeur supérieure ou égale à `seuil` observée depuis le début de cette itération. Initialement, on attribue à `seuil` la valeur 0, et à `min` celle du plus grand entier non signé représentable.

```
void recopie_plus_petits(unsigned t[], unsigned n,
                         unsigned u[], unsigned m)
{
  unsigned i, j, min, seuil;

  for (i = 0, seuil = 0; i < m; i++)
    {
      for (j = 0, min = -1; j < n; j++)
        if (t[j] >= seuil && t[j] < min)
          min = t[j];

      u[i] = min;
      seuil = min + 1;
    }
}
```

   (b) La boucle principale effectue `m` itérations, et chacune d'entre elles parcourt les `n` éléments du tableau `t`. La complexité en temps de la procédure vaut donc $O(\mathtt{m.n})$.

3. (a) Cette fonction retourne 0 si $\mathtt{b} = 0$, et le résultat de la division entière de `a` par `b` sinon.

   (b) Si $\mathtt{b} = 0$, la profondeur de récursion est égale à 1, sinon elle est égale à $(\mathtt{a} \div \mathtt{b}) + 1$, où "$\div$" dénote la division entière. La complexité en espace de la fonction vaut donc $O(\mathtt{a} \div \mathtt{b})$.

(c)
```
unsigned f(unsigned a, unsigned b)
{
  if (b)
    return a / b;
  else
    return 0;
}
```

4. (a)
```
struct noeud
{
  int id;
  struct noeud *succ;
};

struct graphe_cyclique
{
  struct noeud *initial;
};
```

(b)
```
#include <stdlib.h>

struct graphe_cyclique *nouveau_graphe_cyclique(int ids[],
                                                unsigned n)
{
  struct graphe_cyclique *g;
  struct noeud *nds;
  unsigned i;

  g = malloc(sizeof(struct graphe_cyclique));
  if (!g)
    return NULL;

  if (!n)
    {
      g -> initial = NULL;
      return g;
    }

  nds = malloc(n * sizeof(struct noeud));
  if (!nds)
    {
```

```
      free(g);
      return NULL;
    }

  for (i = 0; i < n; i++)
    {
      nds[i].id = ids[i];
      nds[i].succ = nds + (i + 1) % n;
    }

  g -> initial = nds;

  return g;
}
```

(c)
```
#include <stdlib.h>
#include <stdio.h>

void affiche_et_libere(struct graphe_cyclique *g)
{
  struct noeud *n;

  if (g -> initial)
    {
      n = g -> initial;
      do
        {
          printf(" %d", n -> id);
          n = n -> succ;
        }
      while (n != g -> initial);

      printf("\n");
      free(g -> initial);
    }

  free(g);
}
```