Cours d'introduction à l'informatique
Examen de janvier 2023
Énoncés et solutions

Énoncés

1. (a) Écrire une fonction acceptant en arguments deux tableaux de nombres entiers `t1` et `t2` ainsi que le nombre (identique) `n` d'éléments de ces tableaux, et retournant le plus grand indice i pour lequel $\texttt{t1}[i] = \texttt{t2}[i]$. Par exemple, pour les tableaux $[0, -1, 0, 10, 10]$ et $[10, -1, -10, 10, 0]$, cette fonction doit retourner 3. Si la condition $\texttt{t1}[i] = \texttt{t2}[i]$ n'est vraie pour aucun indice i, alors la fonction doit retourner -1.

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

2. L'*indicatrice d'Euler* est une fonction ϕ telle que pour tout nombre entier strictement positif n, $\phi(n)$ a pour valeur le nombre d'entiers m contenus dans l'intervalle $[1, n]$ tels que $\text{pgcd}(m, n) = 1$, où $\text{pgcd}(m, n)$ est le plus grand commun diviseur de m et de n.

   (a) Donner une implémentation de cette fonction, en supposant que l'on dispose d'une bibliothèque qui implémente déjà la fonction pgcd.

   (b) Calculer la complexité en temps de la fonction obtenue au point (a), en sachant que la complexité en temps de $\text{pgcd}(m, n)$, avec $m \leq n$, est $O(\log m)$.

3. On dispose d'un vecteur d'entiers triés par ordre croissant, dans lequel on souhaite rechercher s'il contient ou non un nombre x donné. Pour résoudre ce problème, on utilise l'algorithme suivant :

   — Si le vecteur est vide, alors la réponse est non.

   — Si le vecteur ne contient qu'un seul élément, alors on l'examine afin de déterminer s'il est égal à x.

   — Si le vecteur contient plus d'un élément, alors on le sépare en deux parties plus petites, correspondant à un préfixe et un suffixe du vecteur situés de part et d'autre d'un élément situé approximativement au milieu de celui-ci. Si cet élément n'est pas égal à x, alors on poursuit récursivement

la recherche dans une seule de ces deux parties (devant être choisie judicieusement) à l'aide du même algorithme.

(a) Donner une implémentation (récursive) de cet algorithme, sous la forme d'une fonction retournant une valeur booléenne vraie si le nombre recherché appartient au vecteur, et fausse sinon.

(b) Déterminer les complexités en temps et en espace de la fonction obtenue au point (a).

4. (a) Écrire un fragment de code définissant un type structuré capable de représenter une séquence de mots. Une instance de ce type est composée d'un champ `nb_mots` représentant le nombre de mots qui appartiennent à la séquence, et d'un champ `mots` pointant vers un vecteur de `nb_mots` pointeurs vers des chaînes de caractères, correspondant chacune à un mot de la séquence.

(b) Écrire une fonction prenant en argument une chaîne de caractères, et retournant un pointeur vers une représentation nouvellement allouée de la séquence des mots qui composent cette chaîne. On considère que ces mots sont délimités dans la chaîne par le caractère d'espacement `' '`. Par exemple, la chaîne `"Introduction à l'informatique"` est composée des trois mots `"Introduction"`, `"à"` et `"l'informatique"`. Le nombre de mots et leur longueur ne sont pas connus à l'avance, et peuvent grandement varier d'une chaîne à une autre. En cas d'erreur, la fonction doit retourner un pointeur vide.

Notes :

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

— Il est permis d'employer la fonction `strncpy(dest, src, nb)` de la bibliothèque standard, qui recopie la chaîne de caractères pointée par `src` à l'endroit donné par `dest`, en écrivant au plus `nb` caractères. Le fichier d'en-tête correspondant est `string.h`.

(c) Écrire une fonction permettant de libérer une représentation d'une séquence de mots créée par la fonction obtenue au point (b).

Exemples de solutions

1. (a) Il suffit de parcourir les tableaux depuis leur dernier élément vers le premier, en s'arrêtant dès que l'on trouve un indice pour lequel les deux tableaux possèdent la même valeur.

```
int plus_grand_indice(int t1[], int t2[], unsigned n)
{
  int i;

  for (i = n - 1; i >= 0; i--)
    if (t1[i] == t2[i])
      break;

  return i;
}
```

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

$$\{\texttt{n} \geq 0\}$$

```
for (i = n - 1; i >= 0; i--)
  if (t1[i] == t2[i])
    break;
```

{i = plus grand indice tel que t1[i] = t2[i], ou -1 s'il n'en existe pas}

Après avoir transformé la boucle `for`, ce triplet devient :

$$\{\texttt{n} \geq 0,\ \texttt{i} = \texttt{n} - 1\}$$

```
while (i >= 0)
  {
    if (t1[i] == t2[i])
      break;
    i--;
  }
```

{i = plus grand indice tel que t1[i] = t2[i], ou -1 s'il n'en existe pas}

La condition t1[i] = t2[i] provoque la sortie de la boucle, ce qui revient à dire que t1[i] $\neq$ t2[i] doit être satisfait pour rester dans cette boucle. Cette condition peut être incorporée au gardien de la boucle, ce qui mène au triplet équivalent suivant :

$$\{\texttt{n} \geq 0,\ \texttt{i} = \texttt{n} - 1\}$$

```
while (i >= 0 && t1[i] != t2[i])
  i--;
```

{i = plus grand indice tel que t1[i] = t2[i], ou -1 s'il n'en existe pas}

Montrons par la méthode des invariants que ce triplet est valide. On trouve un invariant de boucle I en caractérisant les opérations effectuées jusqu'à une itération donnée. Un invariant possible est

$$I : \texttt{n} > 0 \text{ et } \texttt{i} \geq -1 \text{ et } \forall j : \texttt{i} < j < \texttt{n} \Rightarrow \texttt{t1}[j] \neq \texttt{t2}[j].$$

Cet invariant exprime qu'avant et après chaque itération de la boucle, les tableaux `t1` et `t2` diffèrent pour tous les indices plus grands que la valeur courante de `i`.

Démontrons à présent que cet invariant est correct.

— Initialement, on a $\texttt{n} \geq 0$ et $\texttt{i} = \texttt{n} - 1$, ce qui satisfait l'invariant. En effet, il n'existe dans ce cas aucun indice j satisfaisant $\texttt{i} < j < \texttt{n}$.

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

$$\{I,\ \texttt{i} \geq 0,\ \texttt{t1}[\texttt{i}] \neq \texttt{t2}[\texttt{i}]\}$$
$$\texttt{i--;}$$
$$\{I\}$$

Montrons que ce triplet est valide, en notant respectivement `x` et $\texttt{x}'$ la valeur d'une variable `x` avant et après l'itération concernée.
On a $\texttt{n}' = \texttt{n}$ et $\texttt{i}' = \texttt{i} - 1$. On a donc bien $\texttt{n}' \geq 0$, et la condition $\texttt{i} \geq 0$ implique $\texttt{i}' \geq -1$. De plus, étant donné que l'on a initialement $\texttt{t1}[j] \neq \texttt{t2}[j]$ pour tout j tel que $\texttt{i} < j < \texttt{n}$, et que l'on présuppose $\texttt{t1}[\texttt{i}] \neq \texttt{t2}[\texttt{i}]$, cette condition est également vraie pour tout j tel que $\texttt{i}' < j < \texttt{n}'$.

— En fin de boucle, on a $\{I,\ \texttt{i} < 0$ ou ($\texttt{i} \geq 0$ et $\texttt{t1}[\texttt{i}] = \texttt{t2}[\texttt{i}]$)$\}$. Cela implique que `i` est bien le plus grand indice tel que $\texttt{t1}[\texttt{i}] = \texttt{t2}[\texttt{i}]$ si cet indice existe, ou est égal à -1 sinon.

2. (a) On sait que $\text{pgcd}(1, n) = 1$ pour tout n, donc on peut commencer l'énumération des valeurs m à partir de 2. On obtient le code suivant :

```
unsigned pgcd(unsigned, unsigned);

unsigned euler(unsigned n)
{
  unsigned e, m;

  for (e = 1, m = 2; m <= n; m++)
```

```
    if (pgcd(m, n) == 1)
      e++;

  return e;
}
```

(b) Le nombre total d'opérations effectuées est borné par

$$\begin{aligned} O(\log 2 + \log 3 + \cdots + \log \mathtt{n}) &\leq O(\log \mathtt{n} + \log \mathtt{n} + \cdots + \log \mathtt{n}) \\ &= O((\mathtt{n} - 1) \log \mathtt{n}) \\ &= O(\mathtt{n} \log \mathtt{n}). \end{aligned}$$

3. (a) La fonction `recherche_dans_vecteur` prend en arguments un vecteur `v`, le nombre `n` d'éléments de celui-ci, et le nombre `x` à rechercher. Elle se base sur une fonction auxiliaire `recherche_dans_sous_vecteur` qui prend en arguments un vecteur `v`, deux indices `a` et `b` et le nombre `x`. Cette fonction auxiliaire a pour but de rechercher `x` dans le sous-vecteur de `v` qui commence à l'indice `a` et qui se termine à l'indice `b`. Ces choix mènent au code suivant :

```
int recherche_dans_sous_vecteur(int v[], unsigned a,
        unsigned b, int x)
{
  int c;

  if (b < a)
     return 0;

  if (a == b)
     return v[a] == x;

  c = a + (b - a) / 2;

  if (v[c] == x)
    return 1;

  if (v[c] > x)
    return recherche_dans_sous_vecteur(v, a, c - 1, x);
  else
    return recherche_dans_sous_vecteur(v, c + 1, b, x);
}
```

```
int recherche_dans_vecteur(int v[], unsigned n, int x)
{
  return recherche_dans_sous_vecteur(v, 0, n - 1, x);
}
```

Note : Calculer `c` en évaluant `c = (a + b) / 2` aurait présenté un risque de dépassement arithmétique dans le cas où `a + b` est grand.

(b) Dans le pire des cas, un appel à `recherche_dans_sous_vecteur` pour traiter un sous-vecteur de taille $m \geq 2$ conduit à un appel récursif de cette même fonction pour traiter un sous-vecteur de taille $m/2$, et ainsi de suite jusqu'à arriver à un des cas de base $m = 0$ ou $m = 1$. La profondeur de récursion est donc égale à $O(\log m)$. Chaque appel récursif effectue $O(1)$ opérations, donc le coût total en temps de la fonction est aussi égal à $O(\log m)$.

On en déduit que la fonction `recherche_dans_vecteur` présente des complexités en temps et en espace toutes deux égales à $O(\log \mathtt{n})$.

4. (a)
```
struct sequence_mots
{
  unsigned nb_mots;
  char **mots;
};
```

(b) Une solution simple consiste à allouer un unique vecteur de caractères contenant tous les mots de la séquence disposés les uns après les autres. La taille de ce vecteur est alors égale au nombre de caractères non blancs de la chaîne fournie en argument, additionné du nombre de mots contenus dans cette chaîne, car chaque mot doit être suivi d'un symbole terminateur qui occupe un caractère. Il faut donc commencer par compter le nombre de mots et le nombre de caractères non blancs contenus dans la chaîne de caractères, avant de pouvoir procéder à l'allocation de mémoire. On obtient par exemple le code suivant :

```
#include <stdlib.h>

struct sequence_mots *decomposer_chaine(char *c)
{
  struct sequence_mots *s;
  char *p;
  unsigned n, i;

  s = malloc(sizeof(struct sequence_mots));
```

```
if (!s)
  return NULL;

/* Comptage du nombre de mots et du nombre n
   de caractères non blancs */

for (p = c, s -> nb_mots = 0, n = 0; *p;)
  if (*p == ' ')
    p++;
  else
    {
      s -> nb_mots++;
      for (n++, p++; *p && *p != ' '; n++, p++);
    }

if (!s -> nb_mots)
  return s;

/* Allocation du tableau et de la chaîne contenant
   tous les mots */

s -> mots = malloc(s -> nb_mots * sizeof(char *));
if (!s -> mots)
  {
    free(s);
    return NULL;
  }

p = malloc(n + s -> nb_mots);
if (!p)
  {
    free(s -> mots);
    free(s);
    return NULL;
  }

/* Copie et indexage de tous les mots */

for (i = 0; *c;)
  if (*c == ' ')
```

```
      c++;
    else
      {
        for (s -> mots[i++] = p; *c && *c != ' '; c++)
          *p++ = *c;

        *p++ = '\0';
      }

  return s;
}
```

Note : Une autre possibilité aurait consisté à recopier la chaîne à l'aide de la fonction `strncpy` vers une chaîne nouvellement allouée, et d'écrire dans cette dernière un caractère terminateur à la fin de chaque mot. Par rapport à cette solution, celle proposée dans ce corrigé présente l'avantage de ne pas gaspiller de mémoire dans le cas où les mots sont séparés par un grand nombre de caractères blancs.

(c)
```
void liberer_sequence_mots(struct sequence_mots *s)
{
  if (s -> nb_mots)
    {
      free(s -> mots[0]);
      free(s -> mots);
    }

  free(s);
}
```

Note : On exploite le fait que dans notre solution, il n'y a qu'une seule chaîne de caractères qui est allouée, et celle-ci est pointée par le premier élément du tableau de mots.