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

Énoncés

1. (a) Écrire une fonction prenant en arguments un tableau d'entiers `t1`, la taille `n` de ce tableau, et un tableau d'entiers `t2` de taille supérieure ou égale à `n`. Cette fonction doit recopier dans le tableau `t2` tous les éléments de `t1` qui sont strictement positifs, dans le même ordre où ils apparaissent dans `t1`. La fonction doit ensuite retourner le nombre d'éléments recopiés.

   Par exemple, si `t1` contient $[-1, 0, 1, 1, 4, 0, 2]$, alors les éléments 1, 1, 4 et 2 doivent être recopiés dans le tableau `t2`, dans cet ordre, et la fonction doit retourner 4.

   (b) Par la méthode des invariants, démontrer que l'opération effectuée par la fonction développée au point (a) est correcte. Démontrer également que cette opération se termine.

2. (a) Écrire une fonction `nfp` calculant le nombre de diviseurs premiers d'un nombre $n > 0$ donné. (*Rappel :* Un nombre $d > 0$ est dit *premier* s'il possède exactement deux diviseurs.) Par exemple, on doit avoir $\texttt{nfp}(150) = 3$, $\texttt{nfp}(7) = 1$ et $\texttt{nfp}(1) = 0$.

   Pour résoudre ce problème, on suppose que l'on dispose déjà d'une fonction `premier(`m`)` qui retourne une valeur booléenne indiquant si son argument $m > 0$ est premier. La complexité en temps de la fonction `premier(`m`)` est $O(1)$. On souhaite que l'implémentation de la fonction `nfp(`n`)` soit raisonnablement efficace, c'est-à-dire que sa complexité en temps soit meilleure que $O(n)$.

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

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

```
unsigned f(char *s, char c)
{
  unsigned r;

  if (!*s)
    return 0;

  r = f(s + 1, c);

  if (*s == c)
    ++r;

  return r;
}
```

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

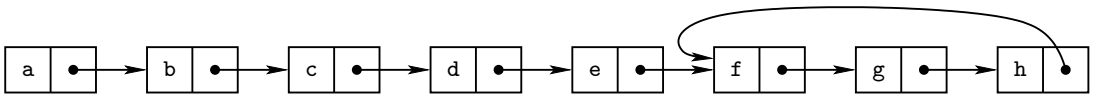
   (c) Écrire une fonction réalisant exactement la même opération que la fonction donnée au point (a), mais sans effectuer d'appel récursif. Pour résoudre ce problème, il n'est pas permis de faire appel à des fonctions issues de la bibliothèque standard.

4. Un outil de manipulation d'automates utilisé pour une application d'informatique théorique doit construire des *structures de lasso*, définies de la façon suivante : Une structure de lasso est caractérisée par deux chaînes de caractères b et p non vides, respectivement appelées *base* et *période*, et dont les longueurs sont notées $\ell(b)$ et $\ell(p)$. La structure est composée de $\ell(b) + \ell(p)$ éléments e_1, e_2, …, $e_{\ell(b)+\ell(p)}$.

   Le premier élément e_1 est appelé l'élément *initial*. Chaque élément contient un caractère et un pointeur. Les caractères contenus dans e_1, e_2, …, $e_{\ell(b)}$ sont ceux de la base b, dans le même ordre que dans cette chaîne. Les caractères contenus dans $e_{\ell(b)+1}$, $e_{\ell(b)+2}$, …, $e_{\ell(b)+\ell(p)}$ sont ceux de la période p, également dans le même ordre.

   Chaque élément e_i tel que $1 \leq i < \ell(b) + \ell(p)$ contient un pointeur vers l'élément suivant e_{i+1}. Le dernier élément $e_{\ell(b)+\ell(p)}$ contient quant à lui un pointeur vers $e_{\ell(b)+1}$, c'est-à-dire vers le premier élément associé à la période.

   Par exemple, le diagramme suivant représente une structure de lasso correspondant à la base `"abcde"` et à la période `"fgh"`. L'élément initial est celui situé à gauche.



(a) Écrire un fragment de code définissant un type structuré pour un élément d'une structure de lasso. *Note :* Vous avez la liberté d'insérer dans les éléments des données supplémentaires de votre choix. La base et la période d'une structure de lasso doivent cependant être uniquement représentées par les caractères figurant dans les éléments de cette structure, et non comme des chaînes de caractères mémorisées séparément.

(b) Écrire une fonction prenant en arguments une base et une période, fournies sous la forme de deux chaînes de caractères b et p. Cette fonction doit retourner un pointeur vers l'élément initial d'une structure de lasso nouvellement allouée, correspondant à cette base et cette période. (Par exemple, appeler cette fonction avec les arguments `"abcde"` et `"fgh"` doit conduire à créer une structure de lasso identique à celle du diagramme précédent.) Le type des éléments de la structure doit bien sûr être celui de votre réponse au point (a).

En cas d'erreur, la fonction doit retourner un pointeur vide. Il est permis de supposer que les chaînes b et p passées en arguments sont non vides. Vous pouvez si vous le souhaitez utiliser la fonction `strlen` de la bibliothèque standard pour calculer la longueur de chaînes de caractères.

(c) Écrire une fonction prenant en argument un pointeur vers le premier élément d'une structure de lasso créée par la fonction développée en réponse au point (b), et libérant la mémoire occupée par cette structure.

Exemples de solutions

1. (a)
```
unsigned recopier(int t1[], unsigned n, int t2[])
{
  unsigned i, j;

  for (i = j = 0; i < n; i++)
    if (t1[i] > 0)
      t2[j++] = t1[i];

  return j;
}
```

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

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

```
for (i = j = 0; i < n; i++)
  if (t1[i] > 0)
    t2[j++] = t1[i];
```

$$\{\texttt{t2}[0 : \texttt{j} - 1] = \text{éléments strictement positifs de } \texttt{t1}[0 : \texttt{n} - 1]\},$$

où $\texttt{t}[a : b]$ dénote la partie du tableau `t` située entre les indices a et b (ces indices étant compris).
En décomposant la boucle `for`, on obtient le triplet équivalent

$$\{\texttt{n} \geq 0,\ \texttt{i} = 0,\ \texttt{j} = 0\}$$

```
while (i < n)
  {
    if (t1[i] > 0)
      t2[j++] = t1[i];
    i++;
  }
```

$$\{\texttt{t2}[0 : \texttt{j} - 1] = \text{éléments strictement positifs de } \texttt{t1}[0 : \texttt{n} - 1]\},$$

On obtient un invariant de boucle I en décrivant le travail effectué par cette boucle jusqu'à une itération donnée. Un invariant possible est

$I : 0 \leq \texttt{i} \leq \texttt{n}$ et $\texttt{t2}[0 : \texttt{j} - 1] =$ éléments strictement positifs de $\texttt{t1}[0 : \texttt{i} - 1]$.

Cet invariant exprime qu'avant et après chaque itération de la boucle, `j` contient le nombre d'éléments strictement positifs du tableau `t1` dont l'indice est strictement inférieur à `i`, et que ces éléments forment les `j` premiers éléments du tableau `t2`.

Montrons maintenant que cet invariant est valide.

— Initialement, on a $\texttt{n} \geq 0$, $\texttt{i} = 0$ et $\texttt{j} = 0$. Les tableaux $\texttt{t1}[0 : \texttt{i} - 1]$ et $\texttt{t2}[0 : \texttt{j} - 1]$ sont tous deux vides, donc l'invariant est satisfait.

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

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

```
if (t1[i] > 0)
  t2[j++] = t1[i];
i++;
```

$$\{I\}$$

Montrons que ce triplet est valide, en notant respectivement $\mathtt{x}$ et $\mathtt{x}'$ la valeur d'une variable $\mathtt{x}$ avant et après l'itération concernée.

— Dans tous les cas, on a $\mathtt{i}' = \mathtt{i} + 1$ qui, combiné avec les éléments de précondition $0 \leq \mathtt{i} < \mathtt{n}$, entraîne $0 < \mathtt{i}' \leq \mathtt{n}$.

— Si $\mathtt{t1}[\mathtt{i}] > 0$, alors on a $\mathtt{j}' = \mathtt{j}+1$. Par la précondition, $\mathtt{t2}[0 : \mathtt{j}-1]$ contient les éléments strictement positifs de $\mathtt{t1}[0 : \mathtt{i} - 1]$. Il en découle que $\mathtt{t2}[0 : \mathtt{j}' - 1] = \mathtt{t2}[0 : \mathtt{j} - 1]; \mathtt{t1}[\mathtt{i}]$ est composé des éléments strictement positifs de $\mathtt{t1}[0 : \mathtt{i}' - 1]$, où ";" dénote la concaténation de deux tableaux.

— Si $\mathtt{t1}[\mathtt{i}] \leq 0$, alors on a $\mathtt{j}' = \mathtt{j}$. On a dans ce cas $\mathtt{t2}[0 : \mathtt{j}' - 1] = \mathtt{t2}[0 : \mathtt{j} - 1]$, qui contient par la précondition les éléments strictement positifs de $\mathtt{t1}[0 : \mathtt{i} - 1]$, et donc aussi ceux de $\mathtt{t1}[0 : \mathtt{i}' - 1]$.

— En fin de boucle, on a $\{I, \mathtt{i} \geq \mathtt{n}\}$, qui implique $\mathtt{i} = \mathtt{n}$. On en déduit que $\mathtt{t2}[0 : \mathtt{j} - 1]$ contient les éléments strictement positifs de $\mathtt{t1}[0 : \mathtt{n} - 1]$, ce qui correspond bien à la postcondition.

On peut montrer que la boucle se termine grâce au variant de boucle

$$v = \mathtt{n} - \mathtt{i},$$

qui est décrémenté à chaque itération, et reste non-négatif comme conséquence de l'invariant I.

2. (a) Chaque diviseur premier de n est égal soit à n, soit à un élément de l'intervalle $[2, \sqrt{n}]$ que l'on peut balayer explicitement. Afin de rendre le code plus efficace, une amélioration consiste à diviser la valeur courante de n par chaque facteur premier trouvé, le plus de fois possible. On obtient le code suivant.

```
unsigned nfp(unsigned n)
{
  unsigned d, nb_div;

  for (nb_div = 0, d = 2; d * d <= n; d++)
    if (!(n % d) && premier(d))
      {
        ++nb_div;
```

```
        do
          n /= d;
        while (!(n % d));
      }

  if (premier(n))
    nb_div++;

  return nb_div;
}
```

(b) Le cas le plus défavorable est celui d'un nombre n premier, pour lequel toutes les valeurs de l'intervalle $[2, \sqrt{n}]$ finissent par être énumérées. Pour chacune d'entre elles, le travail effectué est $O(1)$. On a donc au total une complexité en temps égale à $O(\sqrt{n})$.

3. (a) Cette fonction calcule le nombre de caractères égaux à `c` dans la chaîne pointée par `s`.

(b) La profondeur de récursion est égale à la longueur $|*\,\mathtt{s}|$ de la chaîne de caractères pointée par `s`, et chaque appel récursif consomme une quantité bornée de mémoire. La complexité en espace de la fonction vaut donc $O(|*\,\mathtt{s}|)$.

(c)
```
unsigned f(char *s, char c)
{
  unsigned r;

  for (r = 0; *s; s++)
    if (*s == c)
      r++;

  return r;
}
```

4. (a)
```
struct el_lasso
{
  char c;
  struct el_lasso *next;
};
```

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

struct el_lasso *new_lasso(char *b, char *p)
{
  unsigned len_b, len_p, i;
  struct el_lasso *e;

  len_b = strlen(b);
  len_p = strlen(p);

  if (!len_b || !len_p)
    return NULL;

  e = malloc((len_b + len_p) * sizeof(struct el_lasso));
  if (!e)
    return NULL;

  for (i = 0; i < len_b; i++)
    e[i].c = b[i];

  for (i = 0; i < len_p; i++)
    e[len_b + i].c = p[i];

  for (i = 0; i < len_b + len_p - 1; i++)
    e[i].next = e + i + 1;

  e[i].next = e + len_b;

  return e;
}
```

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

void free_lasso(struct el_lasso *e)
{
  free(e);
}
```