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

Énoncés

1. (a) Écrire une fonction prenant en argument un pointeur vers une chaîne de caractères, et retournant une valeur booléenne indiquant si cette chaîne est entièrement composée de chiffres, à l'exclusion de tout autre type de caractère.

   Par exemple, pour les chaînes `"2024"`, `"007"` et `""`, la fonction doit retourner une valeur vraie. Pour les chaînes `"1 2 3"`, `"INFO2009"` et `"a4"`, elle doit retourner une valeur fausse.

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

2. Le *triangle de Pascal* est une construction mathématique constituée d'une séquence de lignes de longueur croissante : pour $i = 0, 1, 2, \ldots$, la ligne d'indice i contient $i + 1$ nombres entiers C_i^0, C_i^1, $\ldots$, C_i^i, définis de la façon suivante :

   — Le premier et le dernier élément de chaque ligne sont égaux à 1 :

   $$\forall i \geq 0 : \; C_i^0 = C_i^i = 1.$$

   — Si les lignes sont centrées les unes sur les autres, alors chaque élément autre que le premier et le dernier est égal à la somme des deux éléments de la ligne précédente qui sont situés immédiatement à sa gauche et à sa droite :

   $$\forall 0 < j < i : \; C_i^j = C_{i-1}^{j-1} + C_{i-1}^j.$$

   À titre d'exemple, les six premières lignes du triangle de Pascal (correspondant aux indices de 0 à 5) sont les suivantes :

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}$$

(a) Écrire une fonction *pascal* prenant en argument l'indice i d'une ligne du triangle de Pascal, et retournant un pointeur vers un vecteur nouvellement alloué contenant les éléments de cette ligne.
Suggestion : Allouer un vecteur de taille appropriée, et y écrire successivement les lignes d'indice 0, 1, ..., i du triangle.

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

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

```
unsigned f(unsigned long n)
{
    if (n == 0 || n % 2 != 0)
      return 0;

    return 1 + f(n / 2);
}
```

(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 d'itinéraires de randonnée représente un *parcours* sous la forme d'une liste simplement liée dont chaque élément (appelé *lieu*) est composé d'une chaîne de caractères donnant le nom du lieu, ainsi qu'un pointeur vers le prochain lieu à visiter. Un parcours se termine toujours par son point de départ, c'est-à-dire que le dernier élément de la liste doit pointer vers le premier.

(a) Écrire un fragment de code définissant un type structuré capable de représenter un lieu d'un parcours.
Note : Une instance de ce type ne doit pas mémoriser le contenu de la chaîne de caractères décrivant le lieu, mais doit seulement contenir un pointeur vers cette chaîne.

(b) Écrire une fonction prenant en argument une chaîne de caractères, et retournant un pointeur vers une représentation nouvellement allouée du parcours dont les noms de lieux figurent successivement dans la chaîne de caractères, séparés par un espace. Par exemple, pour la chaîne `"B4 B7b Montefiore"`, le parcours doit être composé de trois lieux correspondant à `"B4"`, `"B7b"` et `"Montefiore"`, dans cet ordre.
La fonction doit retourner un pointeur vers le premier lieu de la représentation construite, de même type que votre réponse au point (a), ou un pointeur vide en cas d'erreur.

Notes :

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

— Vous pouvez si vous le souhaitez modifier le contenu de la chaîne fournie en argument, et l'utiliser comme destination des pointeurs contenus dans les éléments de la liste liée.

(c) Écrire une fonction prenant en argument un pointeur vers le premier lieu d'un parcours créé par la fonction du point (b). Cette fonction doit retourner le nombre de lieux qui composent ce parcours, et libérer la représentation de celui-ci.

Exemples de solutions

1. (a)
```
int que_des_chiffres(char *s)
{
  for (; *s; s++)
    if (*s < '0' || *s > '9')
      return 0;

  return 1;
}
```

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

$$\{\mathtt{s}_0 = \mathtt{s}\}$$

```
for (; *s; s++)
  if (*s < '0' || *s > '9')
    return 0;
return 1;
```

{Valeur de retour $\neq 0$ ssi la chaîne $\mathtt{s}_0$ ne contient que des chiffres}

Le code peut être réécrit de façon à faire apparaître explicitement la valeur de retour sous la forme d'une variable fictive `r`. On obtient le triplet équivalent suivant.

$\{\mathtt{s}_0 = \mathtt{s}\}$

```
for (r = 1; *s; s++)
  if (*s < '0' || *s > '9')
    {
      r = 0;
      break;
    }
```

$\{\mathtt{r} \neq 0$ ssi la chaîne $\mathtt{s}_0$ ne contient que des chiffres$\}$

En décomposant la boucle `for` et en remplaçant l'instruction de sortie de boucle `break` par une condition sur `r`, on arrive au triplet suivant.

$\{\mathtt{s}_0 = \mathtt{s}, \mathtt{r} = 1\}$

```
while (*s && r)
  {
    if (*s < '0' || *s > '9')
      r = 0;
    else
      s++;
  }
```

$\{\mathtt{r} \neq 0$ ssi la chaîne $\mathtt{s}_0$ ne contient que des chiffres$\}$

Pour obtenir un invariant de boucle I, on cherche à caractériser les opérations effectuées par la boucle jusqu'à l'itération courante. Un invariant possible est

$$\begin{aligned} I : \exists k \geq 0 : \mathtt{s} = \mathtt{s}_0 + k, & \\ & \mathtt{s}_0[0 : k-1] \text{ ne contient que des chiffres, et} \\ & \text{si } \mathtt{r} = 0, \text{ alors } \mathtt{*s} \neq 0 \text{ et } \mathtt{*s} \text{ n'est pas un chiffre,} \end{aligned}$$

où la notation $\mathtt{s}_0[k_1 : k_2]$ représente la sous-chaîne de $\mathtt{s}_0$ composée des caractères situés entre les indices k_1 et k_2 (ces indices étant compris).

Montrons maintenant que cet invariant est valide.

— Initialement, on a $\mathtt{s} = \mathtt{s}_0$ et $\mathtt{r} = 1$, donc l'invariant I est satisfait (en choisissant $k = 0$).

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

$$\{I,\ \texttt{*s} \neq 0,\ \texttt{r} \neq 0\}$$

```
if (*s < '0' || *s > '9')
  r = 0;
else
  s++;
```

$$\{I\}$$

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. Il y a deux cas à considérer.

— Si `*s` est un chiffre (c'est-à-dire `'0'` $\leq$ `*s` $\leq$ `'9'`), alors on a $\texttt{r}' = \texttt{r} \neq 0$, $\texttt{s}' = \texttt{s} + 1$, et $\texttt{s}_0[0 : k]$ est entièrement composée de chiffres, où k est tel que $\texttt{s} = \texttt{s}_0 + k$. L'invariant I est donc satisfait après l'itération.

— Si `*s` n'est pas un chiffre, alors on a $\texttt{r}' = 0$ et $\texttt{s}' = \texttt{s}$. L'invariant est également satisfait.

— En fin de boucle, on a $\{I,\ \texttt{*s} = 0$ ou $\texttt{r} = 0\}$.

— Si $\texttt{r} = 0$, alors l'invariant implique que la chaîne $\texttt{s}_0$ contient au moins un caractère qui n'est pas un chiffre.

— Si $\texttt{r} \neq 0$, alors `*s` est le caractère terminateur, et l'invariant implique que la chaîne $\texttt{s}_0$ ne contient que des chiffres.

Dans les deux cas, la postcondition est satisfaite.

2. (a) La difficulté principale est de ne pas écraser les éléments de la ligne d'indice $i - 1$ pendant le calcul de ceux de la ligne i. On peut procéder comme pour l'implementation du tri par fusion vue au cours, et se servir d'un vecteur auxiliaire dans lequel on recopie le contenu de la ligne $i-1$ avant de commencer à calculer la ligne i. Une façon simple de réaliser cela consiste à allouer un vecteur de taille $2i+1$, dont les $i+1$ premiers éléments servent au calcul de la ligne d'indice i, et les i éléments suivants constituent le vecteur auxiliaire. À la fin du calcul, on redimensionne ce vecteur de façon à ne pas gaspiller de mémoire.

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

unsigned long *pascal(unsigned i)
{
  unsigned long *v, *v_aux, *r;
  unsigned j, k;

  v = malloc((2 * i + 1) * sizeof(unsigned long));
  if (!v)
    return NULL;

  v_aux = v + i;

  for (j = 0; j <= i; j++)
    {
      if (j)
        memcpy(v_aux, v, j * sizeof(unsigned long));

      v[0] = 1;
      v[j] = 1;

      for (k = 1; k < j; k++)
        v[k] = v_aux[k - 1] + v_aux[k];
    }

  r = realloc(v, (i + 1) *  sizeof(unsigned long));

  if (r)
    return r;
  else
    return v;
}
```

(b) Pour calculer la ligne d'indice i, la fonction calcule successivement les lignes d'indices 0, 1, 2, ..., i, ce qui représente

$$O(1 + 2 + 3 + \cdots + (i + 1)) = O\left(\frac{(i + 1)(i + 2)}{2}\right) = O(i^2)$$

opérations. La quantité de mémoire utilisée est celle du vecteur de taille $2i + 1$, s'ajoutant aux variables de taille constante du programme. La

complexité en espace vaut donc

$$O(2i + 1) = O(i).$$

3. (a) Cette fonction retourne l'exposant de la plus grande puissance de 2 qui divise `n`, ou de façon équivalente, le nombre de zéros situés à la fin de la représentation binaire la plus courte [1] de `n`.

   (b) La complexité en espace de cette fonction correspond à sa profondeur de récursion. Le pire cas est obtenu pour les valeurs de `n` égales à une puissance de deux ; la complexité en espace vaut alors $O(\log \mathtt{n})$.

   (c)
```
unsigned f(unsigned long n)
{
  unsigned r;

  for (r = 0; n && n % 2 == 0; n /= 2)
    r++;

  return r;
}
```

4. (a)
```
struct lieu
{
  char *nom;
  struct lieu *suivant;
};
```

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

static void liberer_parcours(struct lieu *p)
{
  struct lieu *p_suivant;

  for (; p; p = p_suivant)
    {
      p_suivant = p -> suivant;
      free(p);
    }
}
```

1. En considérant que la représentation la plus courte de 0 est vide.

```
struct lieu *creer_parcours(char *s)
{
  struct lieu *premier, *q, **q_precedent;

  for (premier = NULL, q_precedent = &premier; *s; s++)
    {
      q = malloc(sizeof(struct lieu));
      if (!q)
        {
          liberer_parcours(premier);
          return NULL;
        }

      q -> nom = s;
      q -> suivant = NULL;

     *q_precedent = q;
      q_precedent = &q -> suivant;

      while (*s && *s != ' ')
        s++;

      *s = '\0';
    }

  return premier;
}
```

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

unsigned compter_et_liberer_parcours(struct lieu *p)
{
  struct lieu *p_suivant;
  unsigned nb;
```

```
  for (nb = 0; p; p = p_suivant)
    {
      nb++;
      p_suivant = p -> suivant;
      free(p);
    }

  return nb;
}
```