Cours d'introduction à l'informatique
Examen de janvier 2026
Énoncés et solutions

Énoncés

1. Un tableau d'entiers `t` est à *préfixes positifs dominants* si chacun de ses préfixes contient un nombre d'éléments strictement positifs au moins égal à son nombre d'éléments strictement négatifs. En d'autres termes, si $\texttt{t}_{<i}$ désigne le sous-tableau de `t` contenant tous les éléments d'indice strictement inférieur à i, le tableau `t` est à préfixe positif dominant si pour tout i, le nombre d'éléments strictement positifs de $\texttt{t}_{<i}$ est supérieur ou égal au nombre d'éléments strictement négatifs de $\texttt{t}_{<i}$. Par exemple, le tableau $[1, -2, 4, -5]$ et le tableau vide sont à préfixes positifs dominants alors que les tableaux $[-1, 4, 5, 8]$ et $[0, -3, 4]$ ne le sont pas.

   (a) Écrire une fonction prenant en arguments un tableau d'entiers `t` ainsi que sa taille `n`, et qui retourne 1 si `t` est à préfixes positifs dominants et 0 sinon.

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

2. (a) Écrire une fonction prenant en argument une chaîne de caractères `s`, et qui calcule efficacement le nombre de segments de `s` composés d'un et un seul caractère. Un segment est défini comme étant une sous-chaîne non vide qui commence et se termine à des positions déterminées.

   Par exemple, la chaîne `"aaabbca"` contient 11 segments composés d'un seul caractère : 4 correspondent à la sous-chaîne `"a"`, 2 à `"aa"`, 1 à `"aaa"`, 2 à `"b"`, 1 à `"bb"` et 1 à `"c"`. Pour cette chaîne, la fonction doit donc retourner 11.

   Astuce : Le nombre de segments contenus dans une sous-chaîne composée de m caractères vaut $\frac{m(m+1)}{2}$.
   Note : Si nécessaire, vous êtes libre de définir des fonctions auxiliaires de votre choix. Il n'est pas permis en revanche d'utiliser des fonctions issues de la bibliothèque standard.

   (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 suivante :

```
unsigned f(unsigned a, unsigned b)
{
  if (!b)
    return 0;

  if (a < b)
     return a;

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

   (b) Quelle est la complexité en espace de la fonction du point (a) ?

   (c) Rédiger une fonction réalisant exactement la même opération que la fonction du point (a), mais avec des complexités en temps et en espace toutes deux égales à $O(1)$.

4. Un jeu vidéo représente son aire de jeu par une matrice de n lignes et m colonnes, ces valeurs étant supposées non nulles. Chaque élément de cette matrice contient la couleur d'un pixel, composée de trois valeurs entières r, g et b comprises entre 0 et 255.

   (a) Écrire un fragment de code définissant deux types structurés `couleur` et `aire` capables de représenter respectivement la couleur d'un pixel et une aire de jeu de dimension quelconque. Les nombres de lignes et de colonnes d'une aire de jeu doivent faire partie de sa représentation.

   (b) Écrire une fonction prenant en arguments un nombre de lignes, un nombre de colonnes et une couleur, et retournant un pointeur vers une représentation nouvellement allouée d'une aire de jeu possédant ce nombre de lignes et de colonnes, dont tous les pixels sont initialisés à cette couleur. En cas d'erreur, la fonction doit retourner le pointeur nul.

   (c) Écrire une fonction prenant en arguments un pointeur vers la représentation d'une aire de jeu, les indices d'une ligne et d'une colonne (supposés numérotés à partir de 0) et une couleur, et attribuant cette couleur au pixel de l'aire de jeu situé à cette ligne et cette colonne. La couleur des autres pixels de l'aire de jeu reste inchangée.

   (d) Écrire une fonction prenant en arguments un pointeur vers la représentation d'une aire de jeu, et retournant 1 si tous les pixels qu'elle contient sont de la même couleur, et 0 sinon.

Note : Vous pouvez définir des fonctions et types de données supplémentaires de votre choix, ainsi qu'utiliser des fonctions de la bibliothèque standard.

Exemples de solutions

1. (a) Il suffit de considérer tous les préfixes du tableau par ordre croissant de taille, et de compter le nombre d'éléments strictement positifs et strictement négatifs qu'ils contiennent. On obtient le code suivant.

```
int ppd(const int t[], unsigned n)
{
  unsigned i, nb_positifs, nb_negatifs;

  nb_positifs = nb_negatifs = 0;

  for (i = 0; i < n; i++)
    {
      if (t[i] > 0)
        nb_positifs++;
      else
        if (t[i] < 0)
          nb_negatifs++;

      if (nb_negatifs > nb_positifs)
        return 0;
    }

  return 1;
}
```

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

$$\{\mathtt{n} \geq 0\}$$

```
nb_positifs = nb_negatifs = 0;

for (i = 0; i < n; i++)
  {
    if (t[i] > 0)
```

```
      nb_positifs++;
    else
      if (t[i] < 0)
        nb_negatifs++;

    if (nb_negatifs > nb_positifs)
      return 0;
  }

return 1;
```

$$\left\{ \text{valeur retournée} = \left[\begin{array}{ll} 1 & \text{si } \texttt{t} \text{ est à préfixes positifs dominants} \\ 0 & \text{sinon.} \end{array} \right. \right\}$$

En décomposant la boucle `for` et en modifiant le code de façon à ce que la valeur de retour soit attribuée à une variable fictive `r`, on arrive au triplet suivant.

$$\{\texttt{n} \geq 0, \texttt{nb\_positifs} = 0, \texttt{nb\_negatifs} = 0, \texttt{r} = 1, \texttt{i} = 0\}$$

```
while (i < n && r == 1)
  {
    if (t[i] > 0)
      nb_positifs++;
    else
      if (t[i] < 0)
        nb_negatifs++;

    if (nb_negatifs > nb_positifs)
      r = 0;

    i++;
  }
```

$$\left\{ \texttt{r} = \left[\begin{array}{ll} 1 & \text{si } \texttt{t} \text{ est à préfixes positifs dominants} \\ 0 & \text{sinon.} \end{array} \right. \right\}$$

Notons que ce fragment de code n'est pas strictement équivalent au précédent, car il incrémente `i` après avoir affecté la valeur 0 à `r`. Étant donné que cette opération conduit à sortir immédiatement de la boucle,

et que la variable `i` n'est pas impliquée dans la postcondition, cette différence n'a aucun impact sur la validité du triplet.

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 : $0 \leq \texttt{i} \leq \texttt{n}$ et
`nb_positifs` = nombre de valeurs > 0 dans $\texttt{t}[0 : \texttt{i} - 1]$ et
`nb_negatifs` = nombre de valeurs < 0 dans $\texttt{t}[0 : \texttt{i} - 1]$ et
$\texttt{r} = \begin{cases} 1 & \text{si } \texttt{t}[0 : \texttt{i} - 1] \text{ est à préfixes positifs dominants} \\ 0 & \text{sinon,} \end{cases}$

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).

Montrons maintenant que cet invariant est valide.

— Initialement, on a $\texttt{n} \geq 0$, $\texttt{i} = 0$, `nb_positifs` $= 0$, `nb_negatifs` $= 0$ et $\texttt{r} = 1$. L'invariant I est satisfait, puisque le sous-tableau $[0 : -1]$ ne contient aucun élément. Il est donc à préfixes positifs dominants.

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

$\{I,\ \texttt{i} < \texttt{n},\ \texttt{r} = 1\}$

```
if (t[i] > 0)
  nb_positifs++;
else
  if (t[i] < 0)
    nb_negatifs++;

if (nb_negatifs > nb_positifs)
  r = 0;

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 $0 \leq \texttt{i} < \texttt{n}$ et $\texttt{n}' = \texttt{n}$ entraîne $0 \leq \texttt{i}' \leq \texttt{n}'$.

Ensuite, le tableau $\texttt{t}[0 : \texttt{i}' - 1] = \texttt{t}[0 : \texttt{i}]$ contient l'ensemble des éléments de $\texttt{t}[0 : \texttt{i} - 1]$ ainsi que l'élément supplémentaire $\texttt{t}[\texttt{i}]$. Il y a trois cas à considérer :

- Si $\texttt{t}[\texttt{i}] > 0$, alors $\texttt{nb\_positifs}' = \texttt{nb\_positifs} + 1$ et $\texttt{nb\_negatifs}' = \texttt{nb\_negatifs}$.
- Si $\texttt{t}[\texttt{i}] < 0$, alors $\texttt{nb\_positifs}' = \texttt{nb\_positifs}$ et $\texttt{nb\_negatifs}' = \texttt{nb\_negatifs} + 1$.
- Si $\texttt{t}[\texttt{i}] = 0$, alors $\texttt{nb\_positifs}' = \texttt{nb\_positifs}$ et $\texttt{nb\_negatifs}' = \texttt{nb\_negatifs}$.

Dans chacun de ces cas, $\texttt{nb\_positifs}'$ et $\texttt{nb\_negatifs}'$ sont donc bien respectivement égaux aux nombres d'éléments strictement positifs et strictement négatifs de $\texttt{t}[0 : \texttt{i}' - 1]$.

Enfin, la précondition impose que $\texttt{t}[0 : \texttt{i} - 1]$ est à préfixes positifs dominants, ce qui signifie que tous les préfixes de ce tableau (y compris le tableau entier) contiennent au moins autant d'éléments strictement positifs que d'éléments strictement négatifs. Il y a deux cas possibles :

- Si $\texttt{nb\_negatifs}' > \texttt{nb\_positifs}'$, alors $\texttt{t}[0 : \texttt{i}' - 1]$ contient plus d'éléments strictement négatifs que d'éléments strictement positifs, et donc n'est pas à préfixes positifs dominants. On a alors $\texttt{r}' = 0$, et la postcondition est satisfaite.
- Si $\texttt{nb\_negatifs}' \leq \texttt{nb\_positifs}'$, alors le tableau $\texttt{t}[0 : \texttt{i}' - 1]$ et tous ses préfixes contiennent au moins autant d'éléments strictement positifs que d'éléments strictement négatifs. On a alors $\texttt{r}' = 1$ et la postcondition est satisfaite.

— En sortie de boucle, on a $\{I, (\texttt{i} \geq \texttt{n}$ ou $\texttt{r} \neq 1)\}$.

- Si $\texttt{r} = 1$, alors cette condition implique $\texttt{i} \geq \texttt{n}$, qui combiné avec l'invariant I donne $\texttt{i} = \texttt{n}$. Il découle alors de I que $\texttt{t}[0 : \texttt{n}-1] = \texttt{t}$ est à préfixes positifs dominants.
- Si $\texttt{r} \neq 1$, alors l'invariant I implique $\texttt{r} = 0$, $\texttt{i} \leq \texttt{n}$, et le fait que $\texttt{t}[0 : \texttt{i} - 1]$ n'est pas à préfixes positifs dominants. On en déduit que $\texttt{t}[0 : \texttt{n} - 1] = \texttt{t}$ n'est pas non plus à préfixes positifs dominants, puisqu'il contient un préfixe qui ne l'est pas.

Dans ces deux cas, la postcondition est satisfaite.

2. (a) On peut résoudre ce problème en parcourant une seule fois la chaîne `s`. Lors de ce parcours, on lit un caractère c, et puis on épuise toutes les copies de c qui le suivent immédiatement, jusqu'à arriver à un caractère différent de c ou atteindre la fin de la chaîne. Si l'on a lu m occurrences de c, il suffit alors d'ajouter $\frac{m(m+1)}{2}$ au total courant, comme expliqué dans la suggestion accompagnant l'énoncé. On obtient le code suivant.

```
unsigned nb_segments(const char *s)
{
  unsigned m, total;
  char c;

  for (total = 0; *s; s += m)
    {
      for (c = *s, m = 1; s[m] == c; m++);

      total += (m * (m + 1)) / 2;
    }

  return total;
}
```

   (b) La chaîne n'est parcourue qu'une seule fois par la fonction. En d'autres termes, chaque opération figurant dans le code n'est exécutée qu'au plus une fois pour chaque caractère appartenant à la chaîne. La complexité en temps de la fonction vaut donc $O(|\texttt{s}|)$, où $|\texttt{s}|$ dénote la longueur de la chaîne `s`.

3. (a) Cette fonction retourne le reste de la division de `a` par `b`, ou 0 si `b` $= 0$.

   (b)
```
unsigned f(unsigned a, unsigned b)
{
  return b ? a % b : 0;
}
```

4. (a)
```
typedef struct
{
  unsigned char r, g, b;
} couleur;
```

```
typedef struct
{
  unsigned nb_lignes, nb_colonnes;
  couleur *pixels;
} aire;
```

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

aire *nouvelle_aire(unsigned nb_l, unsigned nb_c, couleur c)
{
  aire *a;
  unsigned nb_pixels, i;

  if (!nb_l || !nb_c)
    return NULL;

  a = malloc(sizeof(aire));
  if (!a)
    return NULL;

  nb_pixels = nb_l * nb_c;
  a -> pixels = malloc(nb_pixels * sizeof(couleur));
  if (!(a -> pixels))
    {
      free(a);
      return NULL;
    }

  for (i = 0; i < nb_pixels; i++)
    a -> pixels[i] = c;

  return a;
}
```

(c)
```
void attribuer_couleur(aire *a, unsigned lgn, unsigned col,
                       couleur c)
{
  a -> pixels[lgn * a -> nb_colonnes + col] = c;
}
```

(d)
```
int meme_couleur(aire *a)
{
  unsigned nb_pixels, i;
  couleur c, ci;

  nb_pixels = a -> nb_lignes * a -> nb_colonnes;

  for (c = a -> pixels[0], i = 1; i < nb_pixels; i++)
    {
      ci = a -> pixels[i];
      if (c.r != ci.r || c.g != ci.g || c.b != ci.b)
        return 0;
    }

  return 1;
}
```