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

Énoncés

1. (a) Écrire une fonction prenant en arguments un tableau d'entiers $\mathbf{t}$ et sa taille $\mathbf{n}$, et qui retourne le nombre d'occurrences de la plus grande valeur contenue dans $\mathbf{t}$, c'est-à-dire le nombre de fois où cette valeur apparaît dans le tableau. Par exemple, si le tableau contient $[20, 0, 20, 5]$, alors la fonction doit retourner 2. Si le tableau contient $[1, -2, 3]$, elle doit retourner 1. Si le tableau est vide, elle doit retourner 0.

Note importante : On demande de programmer cette fonction à l'aide d'une seule boucle.

- (b) Par la méthode des invariants, démontrer que la valeur calculée par cette fonction est correcte.
2. (a) Écrire une procédure prenant en arguments deux tableaux d'octets non signés $\mathbf{t}$ et $\mathbf{u}$, leur taille commune $\mathbf{n}$, et un tableau $\mathbf{b}$ de 256 entiers signés. Pour tout $i \in [0, 255]$, la procédure doit placer dans $\mathbf{b}[i]$ une valeur booléenne vraie si $\mathbf{t}$ et $\mathbf{u}$ contiennent tous les deux le même nombre d'octets égaux à i , et fausse sinon. On demande que cette procédure soit la plus efficace possible.
- (b) Calculer la complexité en temps de la procédure obtenue au point (a). Expliquer pourquoi cette complexité est optimale (c'est-à-dire, pourquoi il n'existe pas de solution présentant une meilleure complexité).

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

```
unsigned f(char *s)
{
    if (!*s)
        return 0;

    return f(s + 1) + ((*s == ' ' || *s == '\n') ? 0 : 1);
}
```

- (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. Il n'est pas permis d'utiliser les fonctions de la librairie standard.
4. Une liste simplement liée contient une séquence $e_1; e_2; \dots; e_n$ d'éléments e_i composés chacun d'une valeur entière et d'un pointeur vers l'élément suivant e_{i+1} dans la séquence. Ce pointeur est vide dans le cas où $i = n$.
- (a) Écrire un fragment de code définissant un type structuré pour un élément de liste liée.
  - (b) Écrire une fonction prenant en argument une liste liée ℓ , donnée sous la forme d'un pointeur vers son premier élément, ou d'un pointeur vide dans le cas où ℓ ne contient aucun élément. Cette fonction doit construire une nouvelle liste liée ℓ' en allouant dynamiquement chacun de ses éléments, de façon à ce que ℓ' soit une copie conforme de ℓ . En d'autres termes, ℓ' doit être composée du même nombre d'éléments que ℓ , et les valeurs entières contenues dans ces deux listes doivent correspondre, dans le même ordre. La fonction doit retourner un pointeur vers le premier élément de ℓ' , ou un pointeur vide si ℓ' ne contient aucun élément.
  - (c) Écrire une fonction prenant en argument une liste liée ℓ , donnée de la même façon qu'au point précédent, et retournant la somme des valeurs entières contenues dans les éléments de ℓ . On demande également que cette procédure libère tout l'espace mémoire occupé par les éléments de ℓ , en supposant qu'ils ont été précédemment alloués dynamiquement.

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

Exemples de solutions

1. (a) Il suffit de balayer le tableau en gérant deux variables, l'une retenant la plus grande valeur rencontrée jusqu'à présent, et l'autre le nombre d'occurrences observées de cette valeur. On obtient le code suivant.

```

unsigned nb_plus_grand(int t[], unsigned n)
{
    unsigned i, nombre;
    int plus_grande;

    if (!n)
        return 0;

    nombre = 1;
    plus_grande = t[0];

    for (i = 1; i < n; i++)
        if (t[i] == plus_grande)
            nombre++;
        else
            if (t[i] > plus_grande)
                {
                    nombre = 1;
                    plus_grande = t[i];
                }

    return nombre;
}

```

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

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

```

if (!n)
    return 0;

nombre = 1;
plus_grande = t[0];

for (i = 1; i < n; i++)
    if (t[i] == plus_grande)
        nombre++;
    else
        if (t[i] > plus_grande)
            {
                nombre = 1;
            }

```

```

        plus_grande = t[i];
    }

```

```

return nombre;

```

{Valeur de retour = nombre d'occurrences de la plus grande valeur dans $t[0 : n - 1]$ },

où la notation $t[a:b]$ dénote le sous-tableau de t formé par les éléments dont l'index est compris entre a et b (inclus).

On distingue deux cas. Si $n = 0$, la valeur retournée 0 est correcte. Il reste donc à considérer le cas $n > 0$, correspondant au triplet suivant :

```

    {n > 0}
    nombre = 1;
    plus_grande = t[0];

    for (i = 1; i < n; i++)
        if (t[i] == plus_grande)
            nombre++;
        else
            if (t[i] > plus_grande)
                {
                    nombre = 1;
                    plus_grande = t[i];
                }

```

{nombre = nombre d'occurrences de la plus grande valeur dans $t[0 : n - 1]$ }

En décomposant la boucle `for` et en considérant l'effet des instructions initialisant les variables `nombre` et `plus_grande`, ce triplet devient :

```

    {n > 0, nombre = 1, plus_grande = t[0], i = 1}
    while (i < n)
        {
            if (t[i] == plus_grande)
                nombre++;

```

```

else
  if (t[i] > plus_grande)
  {
    nombre = 1;
    plus_grande = t[i];
  }
  i++;
}

```

{nombre = nombre d'occurrences de la plus grande
valeur dans $t[0 : n - 1]$ }

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

$I : 1 \leq i \leq n$ et $\text{plus\_grande} =$ plus grande valeur dans $t[0 : i - 1]$ et
 $\text{nombre} =$ nombre d'occurrences de plus_grande dans $t[0 : i - 1]$.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 1$, $\text{plus\_grande} = t[0]$ et $\text{nombre} = 1$, donc l'invariant I est satisfait. En effet, $t[0]$ est la plus grande valeur contenue dans le tableau t limité à son premier élément, où elle n'apparaît qu'une seule fois.
- Pour chaque itération de la boucle, on a le triplet

{ I , $i < n$ }

```

if (t[i] == plus_grande)
  nombre++;
else
  if (t[i] > plus_grande)
  {
    nombre = 1;
    plus_grande = t[i];
  }
  i++;

```

{ I }

Montrons que ce triplet est valide, en notant respectivement x et x' la valeur d'une variable x avant et après l'exécution du code. Premièrement, on a $i' = i + 1$, qui combiné avec la précondition $i < n$ et $n' = n$ entraîne $i' \leq n'$. Ensuite, il y a plusieurs cas à considérer :

- Si $t[i] = \text{plus\_grande}$, alors la plus grande valeur contenue dans $t[0 : i' - 1]$ est identique à celle contenue dans $t[0 : i - 1]$. De plus, $t[0 : i' - 1]$ contient une occurrence supplémentaire de cette valeur par rapport à $t[0 : i - 1]$. On a bien dans ce cas $\text{plus\_grande}' = \text{plus\_grande}$ et $\text{nombre}' = \text{nombre} + 1$.
- Si $t[i] > \text{plus\_grande}$, alors la plus grande valeur contenue dans $t[0 : i' - 1]$ est égale à $t[i]$, et la précondition implique que cette valeur n'apparaît pas dans $t[0 : i - 1]$. On en déduit que le nombre d'occurrences de cette valeur dans $t[0 : i' - 1]$ vaut 1. On a bien dans ce cas $\text{plus\_grande}' = t[i]$ et $\text{nombre}' = 1$.
- Si $t[i] < \text{plus\_grande}$, alors la plus grande valeur contenue dans $t[0 : i' - 1]$ est la même que dans $t[0 : i - 1]$, avec un nombre d'occurrences identique dans ces deux sous-tableaux. On a bien dans ce cas $\text{plus\_grande}' = \text{plus\_grande}$ et $\text{nombre}' = \text{nombre}$.
- En sortie de boucle, on a $\{I, i \geq n\}$, qui implique $i = n$. Il découle alors de l'invariant que nombre contient le nombre d'occurrences de la plus grande valeur contenue dans $t[0 : n - 1]$, ce qui correspond bien à la postcondition.

2. (a) Pour résoudre ce problème, on peut utiliser les éléments de b comme des compteurs :

- On initialise $b[i]$ à 0 pour chaque $i \in [0, 255]$.
- Pour tous les éléments e du tableau t , on incrémente $b[e]$. Pour tous les éléments e' du tableau u , on décrémente $b[e']$. À l'issue de cette étape, pour tout $i \in [0, 255]$, $b[i]$ vaut 0 si et seulement si les tableaux t et u contiennent le même nombre d'éléments égaux à i .
- Il reste alors à compléter, pour tout $i \in [0, 255]$, la valeur booléenne contenue dans $b[i]$.

On obtient alors le code suivant.

```

void meme_nombre(unsigned char t[], unsigned char u[],
                 unsigned n, int b[])
{
    unsigned i;

    for (i = 0; i < 256; i++)
        b[i] = 0;

    for (i = 0; i < n; i++)
    {
        b[t[i]]++;
        b[u[i]]--;
    }

    for (i = 0; i < 256; i++)
        b[i] = !b[i];
}

```

- (b) La fonction exécute successivement trois boucles, dont le nombre d'itérations est respectivement égal à 256, n et 256. Les opérations internes à chaque itération de ces boucles s'effectuent en temps constant. La complexité en temps globale vaut donc $O(256 + n + 256) = O(n)$. Cette complexité est optimale, car pour résoudre le problème, il est nécessaire de lire au moins une fois tous les éléments de t et de u .
3. (a) Cette fonction retourne le nombre de caractères de la chaîne pointée par s qui diffèrent du caractère blanc et du retour à la ligne.
- (b) La profondeur de récursion est égale à $|s| + 1$, où $|s|$ dénote la longueur de la chaîne de caractères pointée par s . La complexité en espace de la fonction vaut donc $O(|s| + 1) = O(|s|)$.

(c)

```

unsigned f(char *s)
{
    unsigned r;

    for (r = 0; *s; s++)
        if (*s != ' ' && *s != '\n')
            r++;

    return r;
}

```

4. (a) struct element

```
{
    int valeur;
    struct element *suivant;
};
```

(b) #include <stdlib.h>

```
void libere_liste(struct element *e)
{
    struct element *e_suivant;

    for (; e; e = e_suivant)
    {
        e_suivant = e -> suivant;
        free(e);
    }
}
```

```
struct element *copie_liste(struct element *e)
{
    struct element *e2, *copie, **p;

    for (p = &copie; e; e = e -> suivant)
    {
        e2 = malloc(sizeof(struct element));
        if (!e2)
        {
            *p = NULL;
            libere_liste(copie);
            return NULL;
        }

        *p = e2;
        e2 -> valeur = e -> valeur;
        p = &e2 -> suivant;
    }

    *p = NULL;
    return copie;
}
```

(c) #include <stdlib.h>

```
int somme_et_libere_liste(struct element *e)
{
    int somme;
    struct element *e_suivant;

    for (somme = 0; e; e = e_suivant)
    {
        somme += e -> valeur;
        e_suivant = e -> suivant;
        free(e);
    }

    return somme;
}
```