Correction de l'examen blanc 2025-2026

1. (a) Pour réaliser cette fonction, nous allons itérer sur chaque élément du tableau. Nous allons garder l'élément avec le plus d'occurences et son nombre d'occurences et chaque fois comparer ce nombre avec le nombre d'occurences du nombre actuel. S'il est plus grand, on va remplacer le nombre garder par celui-ci ainsi que le plus grand nombre d'occurences.

```
int f(int *t, unsigned n) {
    unsigned max_occ = 1;
    int nb = t[0];
    unsigned occ = 1;
    for (unsigned i = 0; i < n-1; i++) {
        if (t[i] == t[i+1])
            occ++;
        else
            occ = 1;
        if (occ > max_occ) {
            max_occ = occ;
            nb = t[i];
        }
    }
    return nb;
}
```

   (b) Pour montrer que le programme est correct, on doit montrer que lorsque le programme termine son exécution, les valeurs des variables sont correctes. On souhaite alors établir la validité du triplet suivant :

$$\{n > 0, t = t_0 \dots t_{n-1}, max\_occ = 1, nb = t_0, occ = 1\}$$

```
for (unsigned i = 0; i < n-1; i++) {
    if (t[i] == t[i+1])
        occ++;
    else
        occ = 1;
    if (occ > max_occ) {
        max_occ = occ;
        nb = t[i];
    }
}
```

$$\{nb = \text{ le plus petit nombre de } t \text{ ayant le plus d'occurences}\}$$

   En décomposant la boucle `for` en boucle `while`, on obtient :

$$\{n > 0, t = t_0 \dots t_{n-1}, max\_occ = 1, nb = t_0, occ = 1, i = 0\}$$

```
while (i < n-1) {
    if (t[i] == t[i+1])
        occ++;
    else
        occ = 1;
    if (occ > max_occ) {
        max_occ = occ;
```

```
            nb = t[i];
        }
        i++;
    }
```

$\{nb =$ le plus petit nombre de t ayant le plus d'occurences$\}$

Pour trouver un invariant de boucle I, on caractérise le traitement effectué par la boucle jusqu'à une itération donnée. De plus, I doit être impliqué par la précondition, doit être vrai autant avant qu'après une itération de la boucle `while`, et doit impliquer la postcondition après la dernière itération. Un invariant possible est :

I :$0 \leq i \leq n-1$
$occ =$ nombre d'occurences du nombre t_i dans $t_0 \dots t_i$
$nb =$ plus petit nombre ayant le plus d'occurences dans $t_0 \dots t_i$
$max\_occ =$ nombre d'occurences de nb

Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 0$, $nb = t_0$, $occ = 1$ et $max\_occ = 1$. On a bien que t_0 est le plus petit nombre ayant le plus d'occurences dans t_0 et comme il s'agit d'un seul élément son nombre d'occurences est bien égal à 1.
- Pour chaque itération de boucle, on a le triplet :

$$\{I, i < n-1\}$$

```
if (t[i] == t[i+1])
    occ++;
else
    occ = 1;
if (occ > max_occ) {
    max_occ = occ;
    nb = t[i];
}
i++;
```

$$\{I\}$$

  Montrons que ce triplet est valide. Notons occ et occ', nb et nb', i et i', et max_occ et max_occ' les valeurs des variables avant et après une certaine itération.
  - À chaque itération, nous obtenons que $i' = i + 1$.
  - Si $t_i = t_{i+1}$ alors $occ' = occ + 1$. On a bien trouvé une occurence supplémentaire de t_i on doit incrémenter occ. On obtient bien alors que occ' = nombre d'occurences de t_i dans $t_0 \dots t_i, t_{i+1}$ = nombre d'occurences de $t_{i'}$ dans $t_0 \dots ti'$
  - Si $t_i \neq t_{i+1}$ alors $occ = 1$. t_{i+1} est un nombre que nous n'avons pas encore vu. Comme c'est la première fois qu'on le voit son nombre d'occurences dans le sous-tableau traité est de 1. On a bien que occ' = nombre d'occurences de t_{i+1} dans $t_0 \dots t_i, t_{i+1}$ = nombre d'occurences de $t_{i'}$ dans $t_0 \dots t_{i'}$.
  - Si $occ' > max\_occ$ alors $max\_occ' = occ'$ et $nb' = t_i$. On a trouvé un nombre qui a plus d'occurences que ceux trouvés jusqu'à maintenant on a donc bien que $nb' = t_i = t_{i'}$ et max_occ' = nombre d'occurences de $nb' = t_{i'}$.
- En fin de boucle, on a $\{I, i \geq n-1\}$, ce qui implique que $i = n-1$, on a donc bien que $nb =$ plus petit nombre ayant le plus d'occurences (avec ce nombre d'occurences étant max_occ) dans $t_0 \dots t_{n-1}$ qui est le tableau t.

2. (a) Pour réaliser cette fonction, il va falloir réaliser une boucle qui va trouver tous les diviseurs de n et de les additionner entre eux. Comme nous devons avoir une complexité théorique en temps meilleure que linéaire par rapport à n, nous allons directement prendre les diviseurs d de n ainsi que le diviseur n/d. Ceci nous permettra de ne faire que $\sqrt{n}$ itérations au lieu de n.

```
int deficiant(int n) {
    int sum = 0;
    for (int i = 1; i*i <= n; i++) {
        if (!(n%i)) {
            sum += i;
            if ((n/i != i)&&(n/i != n))
                sum += n/i;
        }
    }
    if (sum < n)
        return 1;
    return 0;
}
```

   (b) Nous allons itérer $\sqrt{n}$ dans la boucle. La complexité théorique en temps est donc $\mathcal{O}(\sqrt{n})$.

3. (a) La fonction retourne la somme des chiffres du nombre v donné en entré.

   (b) La complexité en temps et en espace correspond au nombre de contextes mis sur la pile. Ce nombre de contextes correspond au logarithme en base 10 du nombre v. La complexité en temps et en espace est donc $\mathcal{O}(\log v)$.

   (c) Voici une façon de réécrire cette fonction :

```
unsigned f(unsigned v) {
    unsigned res;
    for (res = 0; v; v/=10)
        res += v%10;
    return res;
}
```

4. Voici le type structuré :

```
typedef struct etape_t {
    char *instr;
    int num;
    struct etape_t *suiv;
} etape;
```

5. Nous allons itérer sur le tableau de chaine de caractères, donner l'adresse du début de la chaine à notre champs instr et allouer chaque fois une nouvelle étape en lui donnant le numéro correspondant à l'indice +1. On met premier à NULL pour couvrir le cas où la taille est 0.

```
etape *f(char **instructions, unsigned n) {
    etape *premier, *prec;
    premier = NULL;
    for (unsigned i = 0; i < n; i++) {
        etape *current;
        current = malloc(sizeof(etape));
        if (!current) {
```

```
            while (premier != NULL) {
                etape *suivant = premier->suiv;
                free(premier);
                premier = suivant;
            }
        }
        current->suiv = NULL;
        current->num = i+1;
        current->instr = instruction[i];
        if (premier == NULL)
            premier = current;
        else
            prec->suiv = current;
        prec = current;
    }
    return premier;
}
```

6. Il suffit de faire quelque chose de similaire au point précédent quand on vérifie que la malloc a fonctionné correctement :

```
void libere(etape *e) {
    for (etape *current = e; current; current = suivant) {
        etape *suivant = current->suiv;
        free(current);
    }
}
```