Correction de l'examen blanc 2024-2025

1. (a) Pour résoudre ce problème, on peut juste itérer élément par élément dans le tableau donné et si l'élément modulo l'entier donné vaut 0, alors l'élément est multiple de cet entier.

```
unsigned nbMult(int t[], unsigned n, int k) {
    unsigned i = 0;

    for (unsigned j = 0; j < n; j++) {
        if (!(t[j]%k))
            i++;
    }

    return j;
}
```

   (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 :

$$\{n \geq 0, k > 0, t = t[0 : n-1], i = 0\}$$

```
for (unsigned j = 0; j < n; j++)
    if (!(t[j]%k))
        i++;
```

$$\{i = \text{ nombres d'éléments de } t[0 : n-1] \text{ qui sont multiples de } k\}$$

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

$$\{n \geq 0, k > 0, t = t[0 : n-1], i = 0, j = 0\}$$

```
while (j < n) {
    if (!(t[j]%k))
        i++;
    j++;
}
```

$$\{i = \text{ nombres d'éléments de } t[0 : n-1] \text{ qui sont multiples de } k\}$$

   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 possibles est :

$$I : 0 \leq i \leq j \leq n$$
$$i = \text{ nombres d'éléments de } t[0 : j-1] \text{ qui sont multiples de } k$$

   Cet invariant exprime que i contient toujours le nombres d'éléments multiples de k dans le sous-tableau actuel représenté par $t[0 : j-1]$

   Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 0$ et $j = 0$. On a bien qu'il y a 0 éléments multiples de k dans le sous-tableau vide.
- Pour chaque itération de la boucle, on a le triplet :

$$\{I, j < n\}$$

```
if (!(t[j])%k)
    i++;
j++;
```

$$\{i\}$$

Montrons que ce triplet est valide. Notons i et i', j et j' les valeurs des variables avant et après une itération spécifique.
Pour chaque itération, on a $j' = j + 1$. Nous avons deux cas à traiter :
  - Si $t[j]\%k = 0$, alors on a $i' = i + 1$. On a trouvé un nouvel élément dans le tableau qui est multiple de k. On a bien alors que $i + 1 =$ nombre de multiples de k dans le sous-tableau $t[0 : j - 1]+$ nombre de multiples de k dans $t[j]$ Ce qui revient à dire : $i' =$ nombre de multiples de k dans le sous-tableau $t[0 : j' - 1]$.
  - Sinon, $i' = i$. On a bien alors que $i =$ nombre de multiples de k dans le sous-tableau $t[0 : j - 1]+$ nombre de multiples de k dans $t[j]$ Ce qui revient à dire : $i' =$ nombre de multiples de k dans le sous-tableau $t[0 : j' - 1]$.
- En fin de boucle, on a $(I, j = n)$, on a donc bien que $i =$ nombre d'éléments multiples de k dans le sous-tableau $t[0 : j - 1] = t[0 : n - 1]$.

Un variant possible est : $v = n - j$

2. (a) Pour résoudre ce problème, nous pouvons diviser le nombre par 10 jusqu'à arriver à 0

```
unsigned nbChiffres(unsigned n) {
    unsigned res;
    for(res = 0; n > 0; n/=10) {
        res++;
    }
    return res;
}
```

   (b) Pour résoudre ce problème, nous pouvons récupérer tous les nombres premiers jusqu'à $\sqrt{n}$ et chaque fois vérifier s'ils divisent n. Si c'est le cas, on récupère le nombre de chiffres du nombre premier et on divise n par celui-là tant que n est un multiple du nombre premier. On récupère le nombre de fois qu'on a divisé (ceci représente la puissance). Si la puissance est > 1 alors il faut calculer le nombre de chiffre de celle-ci. Une fois qu'on sort de la boucle, on aura un n qui vaudra 1 ou un nombre premier. Dans le deuxième cas, il faut de nouveau calculer le nombre de chiffres. Enfin on compare le nombre de chiffres récupéré et le nombre de chiffres de n.

```
unsigned  extra(unsigned n) {
    unsigned n0 = n;
    unsigned nbTot;
    for (unsigned m = 1; nb_premier(m) * nb_premier(m) <= n
                                                    ; m++) {
        if (!(n%nb_premier(m))) {
            nbTot += nbChiffres(nb_premier(m));
            unsigned puissance;
```

```
            for (puissance = 0; !(n%nb_premier(m));
                                n/=nb_premier(m), puissance++);
            if (puissance > 1)
                nbTot += nbChiffres(puissance);
        }
    }
    if (n > 1)
        nbTot += nbChiffres(n);

    return nbTot > nbChiffres(n0) ? 1 : 0;
}
```

(c) Nous divisons à chaque itération le nombre par 10. Le nombre d'itérations sera donc de $\log_{10} n$. La complexité en temps est donc de $\mathcal{O}(\log n)$.

3. (a) Cette fonction va retourner la chaine de caractères. La chaine `informatique` va devenir `euqitamrofni`.

(b) Nous allons effectuer au plus $\frac{n}{2}$ appels récursifs. La complexité en temps est donc $\mathcal{O}(n)$. Pour la complexité en espace, seule la variable c est créée en local donc la complexité sera bornée par la profondeur de récursion, comme la complexité en temps : $\mathcal{O}(n)$.

(c) On peut réécrire cette fonction de manière itérative comme suit :

```
void f(char *s, unsigned n) {
    for (int i = 0, j = n-1; i < j; i++, j--) {
        char c = t[i];
        t[i] = t[j];
        t[j] = c;
    }
}
```

4. (a) On peut définir un tel type de la façon suivante :

```
typedef struct {
    double *pile;
    unsigned capacite;
    unsigned taille;
} zone_travail;
```

(b) i. Une telle fonction peut s'écrire :

```
zone_travail *creer(unsigned n) {
    zone_travail *zone;
    zone = malloc(sizeof(zone_travail));
    if(!zone)
        return NULL;

    zone->capacite = n;
    zone->taille = 0;
    zone->pile = malloc(sizeof(double)*n);

    return zone;
}
```

ii. Une telle fonction peut s'écrire :

```
void liberer(zone_travail *zone) {
    free(zone->pile);
    free(zone);
}
```

iii. Une telle fonction peut s'écrire :

```
void nouveau(zone_travail *zone, double n) {
    if (zone->taille == zone->capacite)
        return;
    zone->pile[zone->taille-1] = n;
    zone->taille++;
}
```

iv. Une telle fonction peut s'écrire :

```
void addition(zone_travail *zone) {
    if (zone->taille <= 1)
        return;

    zone->pile[zone->taille-2] += zone->pile[zone->taille-1]
    zone->taille--;
}
```

v. Une telle fonction peut s'écrire :

```
void multiplication(zone_travail *zone) {
    if (zone->taille <= 1)
        return;

    zone->pile[zone->taille-2] *= zone->pile[zone->taille-1];
    zone->taille--;
}
```

vi. Une telle fonction peut s'écrire (attention qu'une pile ne peut effectuer que des pop, ce qui suit la spécification LIFO):

```
void afficher(zone_travail *zone) {
    for (int i = zone->taille-1; i >= 0; i--) {
        printf("%lf\n%, zone.pile[i]);
    }
}
```