# Réponses-types de l'examen de janvier 2025

1. (a) Il y a plusieurs façons de réaliser cette fonction. On pourrait le faire en une boucle mais cette solution en contiendra 2

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

   (b) La preuve par invariant sera réalisée de manière générale pour les 3 boucles. Pour ça, nous allons utiliser le symbole $\mathcal{Q}$ à la place de $\leq, \geq$.

   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 :

$$\{n > 0, t = t_0, \ldots, t_{n-1}\}$$

```
for (i = 0; i < n - 1; i++)
    if (!(t[i] Q t[i+1]))
        break;
```

{la fonction retourne 1 si t est monotone 0 sinon}

Nous allons légèrement modifier la fonction dans notre invariant en même temps que décomposer la boucle `for` afin de plus facilement exprimer l'invariant.

$$\{n > 0, t = t_0, \ldots, t_{n-1}, i = 0, res = 1\}$$

```
while (i < n - 1){
    if (!(t[i] Q t[i+1]))
        res = 0;
    i++;
}
```

$\{res = 1$ si tous les éléments sont Q les uns à la suite des autres et 0 sinon$\}$

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 :

$$\begin{aligned} I :& n > 0 \\ & 0 \leq i \leq n-1 \\ & res = 1 ssi \forall j \in [0, i-1] t[j] Q t[j+1] \end{aligned}$$

Cet invariant exprime qu'au début et à la fin de chaque itération, $res = 1$ si tous les éléments du sous-tableau d'indices 0 à $i$ sont $\mathcal{Q}$ les uns à la suite des autres (monotones).

Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 0$ et $res = 1$. Le sous-tableau ne contenant qu'un élément est bien montone.
- Pour chaque itération de boucle, on a le triplet :

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

  ```
  if  (!(t[i] Q t[i+1]))
       res = 0;
  i++;
  ```

  $$\{I\}$$

  Montrons que ce triplet est valide, en notant respectivement $i$ et $i'$, ainsi que $res$ et $res'$, la valeur des variables $i$ et $res$ avant et après l'itération concernée.

  - Dans tous les cas, on a $i' = i + 1$. Des contraintes $0 \leq i < n - 1$, on déduit que $0 < i' \leq n - 1$.
  - Si $!(t[i] \mathcal{Q} t[i+1])$, alors $res' = 0$. Nous avons trouvés deux éléments qui se suivent qui violent le prédicat, le (sous-)tableau n'est donc pas montone.
  - Sinon, $res' = res$. Si $res = 0$ le sous-tableau n'est toujours pas monotone car a violé le prédicat déjà dans le sous-tableau précédent. Si $res = 1$, nous avons bien que le sous-tableau actuel est toujours bien montone. Ceci vérifie la postcondition.
- En fin de boucle on a $\{I, i \geq n-1\}$ ce qui implique que $i = n - 1$. On a donc bien que $res = 1$ si `t` est monotone entre les indices 0 et $n - 1$ et $res = 0$ sinon.

2. (a) L'algorithme de cette procédure consiste à d'abord mettre $m$ éléments dans $u$ avant de vérifier si un des éléments restants dans $t$ n'est pas plus petit que l'un des éléments de $u$. Pour vérifier ceci, on parcourt $u$ élément par élément en comparant avec l'élément actuel de $t$. Si cet élément est plus petit, on va remplacer l'élément de $u$ par celui de $t$ et continuer la comparaison avec les autres éléments de $u$ mais cette fois-ci avec l'élément qui vient d'être remplacé.

```
void f(unsigned *t, unsigned n, unsigned *u, unsigned m) {
    for (unsigned i = 0; i < m; i++)
        u[i] = t[i];
    for (unsigned i = m; i < n; i++) {
        unsigned new = t[i];
        for (unsigned j = 0; j < m; j++) {
            if (new < u[j]){
                unsigned tmp = u[j];
                u[j] = new;
                new = tmp;
            }
        }
    }
}
```

   (b) Cette procédure possède une boucle imbriquée dans une seconde en plus d'une boucle en dehors. La première boucle effectuera $m$ itérations. La seconde effectuera $n-m$ itérations et celle à l'intérieure $m$ itérations. Nous avons donc une complexité qui équivaut à $m + (n-m) \times m$. Ceci peut être approximé par $\mathcal{O}(n \times m)$.

3. (a) Cette fonction calcule le résultat de la division entière de $a$ par $b$ si $b \neq 0$. Sinon, elle retourne 0.

   (b) Pour calculer la complexité en espace de cette fonction, nous allons regarder la pile d'exécution. Cette fonction sera appelée un nombre de fois égal à $\frac{a}{b}+1$. La complexité théorique en espace est donc $\mathcal{O}(x)$ où $x = \frac{a}{b}$ si $b \neq 0$.

   (c) Une fonction possible est :

```
unsigned f(unsigned a, unsigned b){
    return (a/b);
}
```

4. (a) Ces structures peuvent s'écrire :

```
typedef struct noeud_t{
    int identifiant;
    struct noeud_t *suivant;
} noeud;
```

```
typedef struct graphe_t{
    noeud *initial;
}
```

(b) Cette fonction peut s'écrire :

```
graphe *f(int *id, unsigned n) {
    graphe *g;
    g = malloc(sizeof(graphe));
    if (!g)
        return NULL;
    noeud *actu;
    for (unsigned i = 0; i < n; i++){
        if (i == 0) {
            noeud *first = malloc(sizeof(noeud));
            if (!first){
                free(g);
                return NULL;
            }
            first->identifiant = id[i];
            first->suivant = first;
            actu = first;
            g->initial = actu;
        }
        else {
            noeud *new;
            new = malloc(sizeof(noeud))
                if(!new) {
                    liberer(g);
                    return NULL;
                }
            new->identifiant = id[i];
            new->suivant = g->initial;
            actu->suivant = new;
            actu = new;
        }
    }
    return g;
}
```

(c) Cette fonction peut s'écrire :

```
void liberer(graphe *g) {
    if(!g)
        return;
    noeud *first = g->premier;
```

```
    if (first) {
        noeud *n = first->suivant;
        while (n != first) {
            noeud *tmp = n;
            n = n->suivant;
            free(tmp);
        }
    }
    if (first)
        free(first);
    free(g);
}
```