Réponses-types de l'examen d'aout 2024

1. (a) On nous demande d'écrire une fonction qui prend 2 arguments en entrée : `t`, un tableau d'entiers qui est trié par ordre croissant, et `n`, sa taille. Cette fonction doit retourner le nombre de valeurs différentes dans ce tableau. L'algorithme consiste donc à parcourir le tableau élément par élément et de comparer l'élément actuel avec l'élément précédent. S'ils sont différents, nous savons que nous avons un élément supplémentaire à ajouter dans notre compteur, sinon, on passe juste aux éléments suivants.

   Une fonction possible est donc :

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

   (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 :

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

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

$$\{res = \text{nombre d'éléments différents dans } t\}$$

   En décomposant la boucle `for`, on obtient :

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

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

$$\{res = \text{nombre d'éléments différents dans } t\}$$

   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 autnt avant qu'après une itération de la boucle *while*, et doit impliquer la postcondition après la dernière itération. Un invariant possible est :

$$\begin{aligned} I : & n > 0 \\ & 0 \leq i \leq n-1 \\ & \forall j \in [0, i] : res = \text{ nombre d'éléments différents dans } t[0:i] \end{aligned}$$

Cet invariant exprime qu'au début et à la fin de chaque itération, `res` continent le nombre d'éléments différents dans `t` entre les indices 0 et i.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 0$ et $res = 1$. Il y a bien 1 élément (différent) dans le sous-tableau composé d'un élément.
- Pour chaque itération de boucle, on a le triplet :

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

```
if (t[i] != t[i+1])
    res++;
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 une 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 $t[i] \neq t[i+1]$, alors $res' = res + 1$ et comme $i' = i + 1$, on a bien un élément différent supplémentaire entre l'indice 0 et $i + 1 = i'$. Ceci vérifie la postcondition.
  - Sinon, $res' = res$. Il n'y a pas d'éléments différents supplémentaires entre l'indice 0 et $i + 1 = i'$. Ceci vérifie également 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 contient le nombre d'éléments différents dans t entre les indices 0 et $i = n - 1$.

(c) Nous allons parcourir tout le tableau une fois élément par élément. La complexité théorique en temps est donc $\mathcal{O}(n)$.

2. (a) Nous avons constamment besoin de la ligne précédente pour calculer l'actuelle. Pour une question d'optimisation d'espace, au lieu de garder l'entièreté du triangle d'Euler, nous allons chaque fois seulement garder la ligne précédente.

```
void f(unsigned n) {
    float *t1, *t2;
    t1 = malloc(sizeof(float)*n);
    t2 = malloc(sizeof(float)*n);
    for (unsigned i = 1; i <= n; i++) {
        t1[0] = 1;
        if(i == 1) {
            printf{"1\n"$};
            t2[0] = 1;
            continue;
        }
        printf("1 ");
        for (unsigned j = 1; j < i - 1; j++) {
            t1[j] = (i-j)*t2[j-1] + (j+1)*t2[j];
            printf("%f ", t1[j]);
        }
        t1[i-1] = 1;
        printf("1\n");
        for (unsigned j = 0; j < i; j++)
            t2[j] = t1[j];
    }
    free(t1);
    free(t2);
}
```

(b) En ce qui concerne la complexité théorique en espace, nous avons créé deux tableaux de n éléments. Nous avons donc une complexité en espace théorique $\mathcal{O}(n)$. Pour la complexité théorique en temps, nous avons une boucle qui itère n fois et deux boucles sont imbriquées, chacune effectuant i itérations maximum. Comme i n'est pas fixé, nous allons prendre le pire cas, où $i = n$. Nous obtenons donc $n \times (n + n)$. La complexité théorique est donc de $\mathcal{O}(n^2)$.

3. (a) Cette fonction retourne la multiplication de i par j.

   (b) Pour calculer la complexité théorique en espace de cette fonction, nous allons regarder la pile d'exécution. La fonction va être appelée j fois si $j < i$ et $i + 1$ fois sinon. Nous pouvons en conclure que la complexité en temps est : $\mathcal{O}(x)$ où $x = min(i, j)$.

   (c)
```
unsigned f(i, j) {
    return (i*j);
}
```

4. (a) Des structures possibles sont :

```
typedef struct element_t{
    char matricule[8];
```

```
    struct element_t *suivant;
}element;

typedef struct liste_t{
    element *premier;
    unsigned nb_elem;
}liste;
```

(b) Une fonction possible est :

```
int f(liste *l, char *m) {
    element *actu = l->premier;
    element *last;
    for (unsigned i = 0; i < l->nb_elem; i++) {
        last = actu;
        unsigned j;
        for (j = 0; j < 8; j++)
            if (actu->matricule[j] != m[j])
                break;
        if (j == 8)
            return 0;
        actu = actu->suivant;
    }
    element *etudiant;
    etudiant = malloc(sizeof(element));
    etudiant->suivant = NULL;
    for (unsigned i = 0; i < 8; i++)
        etudiant->matricule[i] = m[i];
    l->nb_elem++;
    if (l->nb_elem == 1)
        l->premier = etudiant;
    else
        last->suivant = etudiant;

    return 0;
}
```

(c) Une fonction possible est :

```
void liberer(liste *l){
    element *actu = l->premier;
    for (unsigned i = 0; i < l->nb_elem; i++) {
        element *tmp = actu;
        actu = actu->suivant;
        free(tmp);
    }
    free(l);
}
```