Correction des exercices de la session 10

1.
   - `double *tableau[10][20];`
   -
     - $a+1$ s'élève en un pointeur vers le deuxième élément du tableau a
     - $*a$ retourne le premier élément du tableau a, qui vaut 1
     - On en déduit que l'instruction $b[*a] = a + 1;$ place dans la deuxième case du tableau b (index 1) un pointeur vers le deuxième élément de a
     - L'instruction $c = b;$ fait pointer c vers le premier élément de b
     - L'expression $c{+}{+}$ incrémente c, qui pointe alors vers le deuxième élément de b
     - La première application de l'opérateur $*$ retourne donc la valeur du deuxième élément de b, qui est un pointeur vers le deuxième élément de a
     - La seconde application de $*$ retourne alors la valeur du deuxième élément de a, qui est égale à 0. Ce fragment de code affiche donc "0" suivi par un retour à la ligne

2.
   - ```
     typedef struct {
         char nom[20];
         char prenom[20];
     } personne;
     ```
   - .

     ```
     int chaines_egales(char *s1, char *s2){
         for (; *s1 || *s2; s1++, s2++){
             if (*s1 == *s2)
                 continue;
             if (*s1 >= 'A' && *s1 <= 'Z' &&
                                 *s1 + ;a; - 'A' == *s2)
                 continue;
             if (*s2 >= 'A' && *s2 <= 'Z' &&
                                 *s2 + 'a' - 'A' == *s1)
                 continue;
             return 0;
         }
         return 1;
     }

     int personnes_egales(personne *p1, personne *p2){
         return chaines_egales(p1->nom, p2->nom) &&
                   chaines_egales(p1->prenom, p2->prenom);
     }
     ```
     ```

3.
   - Cette fonction retourn ele produit de tous les entiers contenus dans le tableau qui lui est passé en argument, jusqu'à la première valeur nulle (non incluse)
   - Appelons N le nombre d'entiers contenus dans un tableau v, jusqu'à la première valeur nulle (non incluse). En d'autres termes, soit N le plus petit entier positif ou nul tel que $v[N] = 0$
     L'évaluation de $f(v)$ provoquera $N+1$ appels à la fonction f, chacun de ces appels effectuant un nombre borné d'opérations. La complexité en temps de la fonction f vaut donc $\mathcal{O}(N+1) = \mathcal{O}(N)$
   -
```
long f(int *t){
    long r = 1;

    while (*t)
        r *= *t++;

    return r;
}
```

4.
   - .
```
struct element{
    int valeur;
    struct element *suivant;
};
```
   - .
```
#include <stdlib.h>

struct element *construire_tampon(int t[], unsigned nb){
    unsigned i;
    struct element *e, *e_courant, *e_dernier;

    for (i = 0; i < nb; i++){
        e = malloc(sizeof(struct element));
        if (!e)
            return NULL;

        e->valeur = t[nb - 1 - i];

        if (i)
            e->suivant = e_courant;
        else
            e_dernier = e;

        e_courant = e;
```

```
    }

    if (i)
        e_dernier->suivant = e;

    return e;
}
```

- .

```
int somme_tampon(struct element *e){
    struct element *e_courant;
    int somme;

    for (e_courant = e, somme = 0;;){
        somme += e_courant->valeur;
        e_courant = e_courant->suivant;
        if(e_courant == e)
            return somme;
    }
}
```

- .

```
#include <stdlib.h>

void libere_tampon(struct element *e){
    struct element *e_courant, *e_suivant;

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

5. - Le nombre de copies de c situées au début de la chaine peut être compté grâce à une simple boucle parcourant la chaine s, qui se termine lorsqu'on rencontre un caractère différent de c, ou le terminateur qui correspond à un cas particulier de cette condition. On obtient le code suivant

```
unsigned nb_copies(char *s, char c){
    unsigned i;

    for (i = 0; s[i] == c; i++);
```

```
    return i;
}
```

- On souhaite établir la validité du triplet suivant

$$\{c \neq' \backslash 0'\}$$

```
for (i = 0; s[i] == c; i++);
```

$$\{i = \text{nombres de copies de } c \text{ situées au début de } s\}$$

En décomposant la boucle *for*, on obtient le triplet équivalent

$$\{c \neq' \backslash 0', i = 0\}$$

```
while (s[i] == c)
    i++;
```

$$\{i = \text{nombres de copies de } c \text{ situées au début de } s\}$$

Pour trouver un invariant de boucle I, on caractérise les opérations effectuées par la boucle jusqu'à une itération donnée. Un invarant possible est

$$I : c \neq' \backslash 0' \& i \geq 0 \& \forall k \in [0, i-1] : s[k] = c$$

Cet invariant exprie qu'avant et après chaque itération, tous les caractères de s qui ont été lus par les itérations déjà effectuées sont égaux au caractère c
Montrons que cet invariant est valide

- Initialement, on a $c \neq' \backslash 0'$ et $i = 0$ grâce à la précondition, ce qui satisfait I
- Pour chaque itération de la boucle, on a le triplet

$$\{I, s[i] = c\}$$

```
i++
```

$$\{I\}$$

Montrons que ce triplet est valide, en notant respectivement x et x' la valeur d'une variable x avant et après l'itération concernéee.

  * On a $i' = i + 1$. Étant donné que l'invariant implique $i \geq 0$, on a donc bien $i' \geq 0$. De plus, l'invariant implique $\forall k \in [0, i-1] : s[k] - c$. En combinant cela avec la précondition, on obtient $\forall k \in [0, i'01] : s[k] = c$, ce qui entraine que l'invariant est satisfait à l'issue de l'itération

* En fin de boucle, on a $\{I, s[i] \neq c\}$. On a donc $\forall k \in [0, i-1]$: $s[k] = c$, et $s[i] \neq c$, ce qui signifie que i est égal au nombre de copies de c situées au début de s

Il reste à démontrer que la boucle se termine. Il suffit pour cela de considérer le variant de boucle $\ell = i$, où ℓ est la longueur de la chaine s. Étant donné que l'invariant I implique que $s[k]$ est différent du caractère terminateur pour tout k tel que $0 \leq k \leq i$, ce variant possède toujours une valeur entière non négative. De plus, chaque itération de la boucle incrémente i, et diminue donc la valeur du variant