Correction des exercices du chapitre 5

1. Pour résoudre ce problème, il suffit d'itérer sur chaque élément du tableau afin de les additionner avant de diviser le résultat par le nombre d'éléments.

```
double moyenne(double t[], unsigned n) {
    double tot = 0.0;
    for (unsigned i = 0; i < n; i++) {
        tot += t[i];
    }

    return tot/n;
}
```

2. Pour résoudre ce problème, nous allons utiliser la propriété du Triangle de Pascal en calculant toutes les lignes précédentes de n avant de la calculer. À chaque étape, on affichera le coefficient binomial actuel grâce à la fonction `printf` de la librairie `stdio.h`.

```
void coeff(unsigned n) {
    long **c;
    c = malloc((n+1)*sizeof(long*));
    if (!c)
        return;
    for (unsigned i = 0; i <= n; i++){
        c[i] = malloc((n+1)*sizeof(long));
        if (!c[i]){
            for (unsigned j = 0; j < i; j++)
                free(c[j]);
            free(c);
            return;
        }
    }

    for (unsigned i = 0; i <= n; i++){
        c[i][0] = 1;
        printf("\n%ld", c[i][0]);
        c[i][i] = 1;
        for (unsigned j = 1; j < i; j++){
            c[i][j] = c[i-1][j-1] + c[i-1][j];
            printf(" %ld%, c[i][j]);
        }
        if(i != 0)
            printf(" %ld", c[i][i]);
    }

    for (unsigned i = 0; i <= n; i++)
        free(c[i]);
    free(c);
}
```

3. Une fonction résolvant ce problème est :

```
unsigned max_seq(int t[], unsigned n) {
```

```
    if (n == 0)
        return 0;

    unsigned max = 1;
    unsigned current = 1;

    for (unsigned i = 1; i < n; i++){
        if (t[i-1] == t[i]+1) {
            current++;
            if (current > max)
                max = current;
        }
        else
            current = 1;
    }

    return max;
}
```

4. Pour résoudre ce problème, il suffit d'itérer sur chaque caractère des deux chaines en même temps et retourner l'indice actuel dès que deux caractères sont différents. Si on arrive à la fin des deux chaines en même temps, on peut retourner 0.

```
unsigned pos_diff(char *s1, char *s2) {
    unsigned i;

    for (i = 1; *s1 || *s2; s1++, s2++, i++) {
        if (*s1 != *s2)
            return i;
    }

    return 0;
}
```

5. Pour résoudre ce problème, nous devons tout d'abord calculer la taille de la chaine. Une fois la taille récupérée, nous comparerons le caractère le plus à gauche avec le caractère le plus à droite jusqu'à arriver au caractères les plus au milieu de la chaine.

```
unsigned palindrome(char *s) {
    unsigned taille;
    for (taille = 0; s[taille]; taille++);
    taille -= 1;

    for (unsigned i = 0; i < taille; i++, taille--){
        if (s[i] != s[taille])
            return 0
    }

    return 1;
}
```

6. Pour résoudre ce problème, il suffit de commencer par le dernier caractère des deux chaines puis d'itérer caractère par caractère en arrière jusqu'à arriver à la fin de la chaine ou quand 2 caractères sont différents.

```
unsigned suffixe(char *s1, char *s2) {
    unsigned taille_s1;
    unsigned taille_s2;

    for (taille_s1 = 0; s1[taille_s1]; taille_s1++);
    for (taille_s2 = 0; s2[taille_s2]; taille_s2++);

    unsigned i;
    for (i = 0; (i < taille_s1) && (i < taille_s2) &&
                s1[taille_s1 - 1 - i] == s2[taille_s2 - 1 -i]; i++);

    return i;
}
```

7. Pour résoudre ce problème, il suffit d'itérer caractère par caractère dans les deux chaines en même temps. Dès qu'on tombe sur un espace, on itère juste sur cette chaine jusqu'à arriver au prochain caractère qui n'est pas un espace.

```
unsigned egaux(char *s1, char *s2) {
    while (*s1 || *s2){
        while (*s1 == ' ')
            s1++;
        while (*s2 == ' ')
            s2++;
        if (*s1 != *s2)
            return 0;
    }

    return 1;
}
```

8. (a) Pour résoudre ce problème, il suffit de parcourir les tableaux et de retourner 0 dès qu'on trouve 2 éléments différents.

```
unsigned diff(int *t1, int *t2, unsigned l) {
    for (unsigned i = 0; i < l; i++) {
        if (t1[i] != t2[i])
            return 0;
    }

    return 1;
}
```

   (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 :

$$\{t1 = [t1_0, \ldots, t1_{l-1}], t2 = [t2_0, \ldots, t2_{l-1}], l > 0\}$$

```
for (unsigned i = 0; i < l; i++) {
    if (t1[i] != t2[i])
        return 0;
}
```

{la valeur retournée est 0 si t1 et t2 sont différents, 1 sinon}

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

$$\{t1 = [t1_0, \ldots, t1_{l-1}], t2 = [t2_0, \ldots, t2_{l-1}], l > 0, i = 0\}$$

```
while (i < l) {
    if (t1[i] != t2[i])
        return 0;
    i++;
}
```

{la valeur retournée est 0 si t1 et t2 sont différents, 1 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 :

$$\begin{aligned} I : & 0 \leq i \leq l \\ & \forall 0 \leq k < i : t1[k] = t2[k] \end{aligned}$$

Cet invariant exprime que tous les éléments précédent l'élément d'indice i sont égaux un à un.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 0$. On a bien $i \leq l$ et que tous les éléments avant le premier élément sont égaux un à un.
- Pour chaque itération de la boucle, on a le triplet :

$$\{I, i < l\}$$

```
    if (t1[i] != t2[i])
    return 0;
i++;
```

$$\{I\}$$

Montrons que ce triplet est valide. Notons i, i' la valeur de i avant et après une itération spécifique.
Nous avons 2 cas à traiter :

- Si t1[i] = t2[i], alors $i' = i + 1$. Nous avons donc que $\forall 0 \leq k \leq i : t1[k] = t2[k]$ ce qui revient à $\forall 0 \leq k < i' : t1[k] = t2[k]$.
- Si t1[i] $\neq$ t2[i], alors $i' = i$ et nous quittons la fonction, on a bien que $\forall 0 \leq k < i : t1[k] = t2[k]$ ce qui revient à $\forall 0 \leq k < i' : t1[k] = t2[k]$.

- En fin de boucle, on a $(I, i = l)$ ou $(I, t1[i] \neq t2[i])$. On a bien alors que la fonction retourne 0 si pour un i, on a t1[i] *neq* t2[i] et on quitte la boucle sans return dans le cas où $\forall i < n$: $t1[i] = t2[i]$.

9. (a) Pour résoudre ce problème, il suffit d'itérer caractère par caractère sur les deux chaines en même temps, jusqu'à ce qu'on tombe en fin de chaine ou que 2 caractères ne sont pas identiques.

```
unsigned prefixe(int *t1, int *t2, unsigned l) {
    unsigned i;

    for (i = 0; i < l && t1[i] == t2[i]; i++);

    return i;
}
```

   (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 :

$$\{t1 = [t1_0, \ldots, t1_{l-1}], t2 = [t2_0, \ldots, t2_{l-1}], l > 0\}$$

```
for ( i = 0; i < l && t1[i] == t2[i]; i++);
```

$$\{i = \text{longueur du plus grand préfixe commun}\}$$

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

$$\{t1 = [t1_0, \ldots, t1_{n-1}], t2 = [t2_0, \ldots, t2_{n-1}], l > 0, i = 0\}$$

```
while (i < l && t1[i] == t2[i])
    i++;
```

$$\{i = \text{longueur du plus grand préfixe commun}\}$$

   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 :

$$\begin{aligned} I :& 0 \leq i \leq l \\ & \forall 0 \leq k < i : t1[k] = t2[k] \end{aligned}$$

   Cet invariant exprime que tous les éléments précédent l'élément d'indice i sont égaux un à un.

   Montrons maintenant que cet invariant est valide.

   - Initialement, on a $i = 0$. On a bien $i \leq l$ et que tous les éléments avant le premier élément sont égaux un à un. Le plus grand préfixe commun de deux chaines vides est donc de taille 0.
   - Pour chaque itération de la boucle, on a le triplet :

$$\{I, i < l, t1[i] == t2[i]\}$$

```
i++;
```

$\{I\}$

Montrons que ce triplet est valide. Notons i, i' la valeur de i avant et après une itération spécifique.
Pour chaque itération, on a t1[i] = t2[i] et $i < l$, alors $i' = i + 1$. Nous avons donc que $\forall 0 \leq k \leq i : t1[k] = t2[k]$ ce qui revient à $\forall 0 \leq k < i' : t1[k] = t2[k]$ et on a $i < l$ donc $i' \leq l$.

- En fin de boucle, on a $(I, i = l)$ ou $(I, t1[i] \neq t2[i])$. On a bien alors que $\forall 0 \leq 0 < i : t1[k] = t2[k]$ et donc que i correspond au premier indice où t1 $\neq$ t2, ce qui correspond à la taille du plus grand préfixe commun.

Un variant possible est : $v = l - i$.

10. (a) Pour résoudre ce problème, on va commencer par initialiser tous les éléments du tableau h à 0 puis incrémenter l'élément d'indice correspond à l'élément récupéré dans le tableau t dans une seconde boucle.

```
void count(unsigned char *t, unsigned n, int *h) {

    for (unsigned i = 0; i < 256; i++)
        h[i] = 0;

    for (unsigned i = 0; i < n; i++)
        h[t[i]] += 1;
}
```

(b) Nous réalisons deux boucles. La première a une taille fixe de 256 que nous pouvons donc considéré comme constante alors que la deuxième itère sur le tableau de taille n. La complexité théorique est donc $\mathcal{O}(n)$.

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

$$\{t = [t_0, \ldots, t_{n-1}], h = [0, \ldots, 0], n \geq 0\}$$

```
for (unsigned i = 0; i < n; i++)
    h[t[i]] += 1;
```

$$\{\forall 0 \leq k \leq 255 : h[k] = \text{ nombre de valeurs de } t_0, \ldots, t_{n-1} \text{ égales à } k\}$$

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

$$\{t = [t_0, \ldots, t_{n-1}], h = [0, \ldots, 0], n \geq 0, i = 0\}$$

```
while (i < n) {
    h[t[i]] += 1;
    i++;
}
```

$$\{\forall 0 \leq k \leq 255 : h[k] = \text{ nombre de valeurs de } t_0, \ldots, t_{n-1} \text{ égales à } k\}$$

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 possible est :

$$I : 0 \leq i \leq n$$
$$\forall 0 \leq k < 255 : h[k] = \text{ nombre de valeurs de } t_0, \ldots, t_{i-1} \text{ égales à } k$$

Cet invariant exprime que les éléments d'indice k du tableau h correspondent au nombre de fois que cet élément est présent dans le sous-tableau $t_0 \ldots t_{i-1}$.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 0$. On a bien $i \leq n$ et comme nous traitons un sous-tableau vide tous les éléments de h sont à 0.
- Pour chaque itération de la boucle, on a le triplet :

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

```
h[t[i]] += 1;
i++;
```

$$\{I\}$$

Montrons que ce triplet est valide. Notons i, i' la valeur de i avant et après une itération spécifique ainsi que h et h' le tableau avant et après une itération spécifique.
Pour chaque itération, on a $i' = i + 1$ et $h'[t[i]] = h[t[i]] + 1$. On a bien que comme $i < n$, $i' \leq n$ et comme on incrémente uniquement l'élément de h qui a comme indice l'élément d'indice i de t, que $\forall 0 \leq k < 255 : h[k] =$ nombre de valeurs de $t_0, \ldots, t_{i'-1}$ égales à k.

- En fin de boucle, on a $(I, i = n)$. On a bien alors que $\forall 0 \leq i < n : t[i] = \sum_{j=0}^{i} t[i]$.
- En fin de boucle, on a $(I, i = n)$, on a donc bien que $\forall 0 \leq k < 255 : h[k] =$ nombre de valeurs de $t_0, \ldots, t_{n-1}$ ég

11. (a) On peut itérer sur le tableau et chaque fois ajouter l'élément précédent à l'actuel. Comme chaque élément précédent aura été traité avant l'actuel, on aura chaque fois dans l'élément précédent la valeur que dois avoir l'élément actuel moins ce dernier.

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

(b) Nous parcourons le tableau élément par élément. La complexité en temps de la fonction est donc $\mathcal{O}(n)$.

(c) 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 :

$$\{t = [t_0, \ldots, t_{n-1}], n \geq 0\}$$

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

$$\{t = [t_0, t_0 + t_1, \ldots, t_0 + t_1 + \ldots + t_{n-1}]\}$$

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

$$\{t = [t_0, \ldots, t_{n-1}], n \geq 0, i = 1\}$$

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

$\{t = [t_0, t_0 + t_1, \ldots, t_0 + t_1 + \ldots + t_{n-1}]\}$

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 :

$$I : 0 \leq i \leq n$$

$$\forall 0 \leq k < i : t[k] = \sum_{j=0}^{k} t_0[j] \text{ où } t_0 \text{ correspond au tableau initial}$$

Cet invariant exprime que tous les éléments précédent l'élément d'indice i sont égaux à la somme d'eux-même avec les éléments précédents.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 1$. On a bien $i \leq n$ et $\forall 0 \leq k < 1$ (donc pour $k = 0$) $t[0] = t_0[0] = \sum_{j=0}^{k} t_0[k]$.
- Pour chaque itération de la boucle, on a le triplet :

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

```
t[i] += t[i-1];
i++;
```

$$\{I\}$$

Montrons que ce triplet est valide. Notons i, i' la valeur de i avant et après une itération spécifique ainsi que t et t' le tableau avant et après une itération spécifique.
Pour chaque itération, on a $i' = i + 1$ et $t'[i] = t[i-1] + t[i]$. Comme $t[i-1] = \sum_{j=0}^{i-1} t_0[j]$, on a bien que $t'[i] = \sum_{j=0}^{i} t[j]$ et donc que $\forall 0 \leq k \leq i : t'[k] = \sum_{j=0}^{k} t_0[j] \Leftrightarrow \forall 0 \leq k < i' t'[k] = \sum_{j=0}^{k} t_0[j]$.

- En fin de boucle, on a $(I, i = n)$. On a bien alors que $\forall 0 \leq i < n : t[i] = \sum_{j=0}^{i} t[i]$.

12. (a) Pour résoudre ce problème, nous allons parcourir toute le chaine de caractères et additionner chaque fois que le caractère est compris entre `'0'` et `'9'`.

```
unsigned sum_chiffres(char *s) {
    unsigned tot;

    for (tot = 0; *s; s++)
        if((*s >= '0') && (*s <= '9'))
            tot += *s - '0';

    return tot;
}
```

(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 :

$$\{s = s_0\}$$

```
for (tot = 0; *s; s++)
    if ((*s >= '0') && (*s <= '9'))
        tot += *s - '0';
```

$$\{tot = \text{ total des chiffres contenus dans la chaine } s_0\}$$

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

$$\{s = s_0, tot = 0\}$$

```
while (*s) {
    if ((*s >= '0') && (*s <= '9'))
        tot += *s - '0';
    s++;
}
```

$$\{tot = \text{ total des chiffres contenus dans la chaine } s_0\}$$

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 :

$I : s \geq s_0$
$tot = \text{ total des chiffres contenus dans la sous-chaine } s_0, s_0 + 1, \ldots, s - 1$

Cet invariant exprime que tot va contenir la somme de tous les chiffres contenus dans la sous-chaine actuellement parcourue.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $s = s_0$ et $tot = 0$. Le total des chiffres présents dans la sous-chaine vide est bien de 0.
- Pour chaque itération de la boucle, on a le triplet :

$$\{I, *s\}$$

```
if ((*s >= '0') && (*s <= '9'))
    tot += *s - '0';
s++;
```

$$\{I\}$$

Montrons que ce triplet est valide. Notons tot et tot', s et s' les valeurs des variables avant et après une itération spécifique.
Pour chaque itération, on a $s' = s + 1$.
Nous avons deux cas :

- Si *s est compris entre `'0'` et `'9'`, on aura $tot' = tot +$ *s transformé en int. On a donc bien que $tot =$ total des chiffres présents dans la sous-chaine $s_0, s_1, \ldots, s = s_0, s_1, \ldots, s' - 1$.
- Sinon, on aura $tot' = tot$ et on a bien aussi que $tot =$ total des chiffres présents dans la sous-chaine $s_0, s_1, \ldots, s = s_0, s_1, \ldots, s' - 1$.

• En fin de boucle, on a $(I, !*s)$ et la sous-chaine parcourue correspond à la chaine initiale en entier. On a donc bien que $tot =$ total des chiffres contenus dans la chaine s_0.

13. (a) Pour résoudre ce problème, on va parcourir le tableau et chaque fois que deux éléments consécutifs sont identiques, nous allons le comparer au plus grand nombre trouvé et si le nouvel élément est plus grand, on va remplacer cette variable par cet élément.

```
unsigned max_cons(unsigned *t, unsigned n) {
    unsigned max = 0;

    for (unsigned i = 1; i < n; i++) {
        if ((t[i-1] == t[i])&&(t[i] > max))
            max = t[i];
    }

    return max;
}
```

(b) On parcourt le tableau élément par élément. La complexité en temps est donc $\mathcal{O}(n)$. Nous gérons seulement un tableau de taille n. La complexité en espace est donc $\mathcal{O}(n)$.

14. Pour résoudre ce problème, on parcourt le tableau de chaine de caractères. Pour chaque chaine, on regarde si son premier caractère est le caractère de terminaison ou pas. Si c'est le cas, le chaine est vide.

```
void vide(char **t, unsigned *o, unsigned n) {
    for (unsigned i = 0; i < n; i++)
        o[i] = t[i][0] == '\0';
}
```

15. (a) Pour résoudre ce problème, on va parcourir les deux tableaux et dès que deux éléments au même indice sont identiques, nous allons modifier l'indice gardé.

```
int big_indice_com(int *t1, int *t2, unsigned n) {
    int res = -1;

    for (unsigned i = 0; i < n; i++)
        if(t1[i] == t2[i])
            res = i;

    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 :

$$\{t1 = [t1_0, \ldots, t1_{n-1}], t2 = [t2_0, \ldots, t2_{n-1}], n > 0, res = -1\}$$

```
for (unsigned i = 0; i < n; i++)
    if (t1[i] == t2[i])
        res = i;
```

$\{res =$ le plus grand indice où $t1 = t2\}$

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

$$\{t1 = [t1_0, \ldots, t1_{n-1}], t2 = [t2_0, \ldots, t2_{n-1}], n > 0, i = 0, res = -1\}$$

```
while (i < n) {
    if (t1[i] == t2[i])
        res = i;
    i++;
}
```

$\{res =$ le plus grand indice où $t1 = t2\}$

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 :

$I : 0 \leq i \leq n$
$res =$ plus grand indice tel que $t1 = t2$ dans les sous-tableaux $t1_0, \ldots, t1_{i-1} et t2_0, \ldots, t2_{i-1}$

Cet invariant exprime que res vaut toujours le plus grand indice dans le sous-tableau parcouru.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 0$ et $res = -1$. Les deux sous-chaines sont vides, aucun indice ne peut être le plus grand.
- Pour chaque itération de la boucle, on a le triplet :

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

  ```
  if (t1[i] == t2[i])
      res = i;
  i++;
  ```

  $$\{I\}$$

  Montrons que ce triplet est valide. Notons res et res', i, i' les valeurs des variables avant et après une itération spécifique.
  Pour chaque itération, on a $i' = i + 1$.
  Nous avons deux cas :
  - Si $t1[i] == t2[i]$, alors nous avons que $res' = i$ et i est bien l'indice le plus grand où deux éléments sont identiques dans les sous-tableaux $t1_0, \ldots, 1_{i'-1}$ et $t2_0, \ldots, t2_{i'-1}$.
  - Sinon, $res' = res$ et on a bien aussi que le précédent plus grand indice est toujours le plus grand dans les sous-tableaux $t1_0, \ldots, 1_{i'-1}$ et $t2_0, \ldots, t2_{i'-1}$.
- En fin de boucle, on a $(I, i = n)$. On a donc bien que res contient le plus grand indice où deux éléments sont identiques dans les sous-tableaux $t1_0, \ldots, t1_{n-1}$ et $t2_0, \ldots, t2_{n-1}$.

16. (a) Pour résoudre ce problème, il suffit de d'abord passer tous les espaces en début de chaine grâce à une première boucle, puis de compter le nombre de caractères dans le sous-chaine restante.

```
unsigned longueur(char *s) {
    unsigned i;
    while (*s == ' ')
        s++;
    for (i = 0; *s; i++, s++);

    return i;
}
```

(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 :

$\{s = s_1$, où s_1 correspond à la chaine de caractères après la première boucle$\}$

```
for (i = 0; *s; i++, s++);
```

$\{i =$ nombre de caractères de $s_1\}$

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

$\{s = s_1$, où s_1 correspond à la chaine de caractères après la première boucle, $i = 0\}$

```
while (*s;){
    i++;
    s++;
}
```

$\{i =$ nombre de caractères de $s_1\}$

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 :

$$\begin{aligned} I :& \exists k \geq 0 : s = s_1 + k \\ & i = \text{ longueur de la sous-chaine } s_1[0 : k-1] \\ & s_1[0 : k-1] \text{ ne contient aucun caractère terminateur} \end{aligned}$$

Cet invariant exprime que la valeur de i correspond à la longueur de la sous-chaine $s_1[0 : i-1]$

Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 0$ et $s = s_1$. Le nombre de caractères dans le sous-tableau $s_1[0 : -1]$ est bien de 0.
- Pour chaque itération de la boucle, on a le triplet :

$$\{I, *s\}$$

```
i++;
s++;
```

$$\{I\}$$

Montrons que ce triplet est valide. Notons i et i', s et s' les valeurs des variables avant et après une itération spécifique.
Pour chaque itération, on a $i' = i + 1$ et $s' = s + 1$. On a donc bien que i' est la taille de la sous chaine entre s_1 et $s' - 1$.

- En fin de boucle, on a $(I, !*s)$. On a alors que $i =$ taille de la sous-chaine comprise entre s_1 et s.

Un variant possible est : $v = \ell_1 - i$ où ℓ_1 est la taille de la sous-chaine s_1.

17. (a) Pour résoudre ce problème, il suffit de parcourir la chaine tant que le caractère actuel est égal au caractère recherché.

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

    for (i = 0; *s == c; s++, i++);

    return i;
}
```

(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 :

$$\{s = s_0\}$$

```
for (i = 0; *s == c; s++, i++);
```

$$\{i = \text{ nombre de caractères égaux à c en début de chaine } s_0\}$$

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

$$\{s = s_0, i = 0\}$$

```
while (*s == c) {
    s++;
    i++;
}
```

$$\{i = \text{ nombre de caractères égaux à c en début de chaine } s_0\}$$

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 :

$I : \exists k \geq 0 : s = s_0 + k$
$i =$ longueur de la sous-chaine $s_0[0 : k - 1]$ où $\forall 0 \leq j < k : s_0[j] = c$

Cet invariant exprime que la valeur de i correspond à la longueur de la sous-chaine $s_0[0 : k-1]$ où tous les caractères dans cette sous-chaine sont égaux à c.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 0$ et $s = s_0$. Le nombre de caractères égaux à c dans la sous-chaine vide est bien 0.
- Pour chaque itération de la boucle, on a le triplet :

$$\{I, *s == c\}$$

```
s++;
i++;
```

$$\{I\}$$

Montrons que ce triplet est valide. Notons i et i', s et s' les valeurs des variables avant et après une itération spécifique.
Pour chaque itération, on a $i' = i + 1$ et $s' = s + 1$. On a donc bien que i' est la taille de la sous chaine entre s_0 et s (ou $s' - 1$) et comme on est dans la boucle, on a bien que tous les caractères à l'intérieur sont égaux à c.

- En fin de boucle, on a $(I, *s \neq c)$. On a alors que $i =$ longueur de la sous-chaine de caractères entre s_0 et $s - 1$, ce qui correspond à la sous-chaine ne contenant que des caractères c en debut de chaine.

Un variant possible est : $v = \ell_0 - i$.

18. (a) Pour résoudre ce problème, on doit parcourir le tableau t1 et à chaque fois que l'élément est positif, on va le mettre dans t2 avant d'avancer l'indice de t2.

```
unsigned copie(int *t1, int *t2, unsigned n) {
    unsigned j = 0;

    for (unsigned i = 0; i < n; i++)
        if (t1[i] > 0) {
            t2[j] = t1[i]
            j++;
        }

    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 correcetes. On souhaite alors établir la validité du triplet suivant :

$$\{n \geq 0, j = 0\}$$

```
for (unsigned i = 0; i < n; i++)
    if (t1[i] > 0) {
        t2[j] = t1[i];
        j++;
    }
```

$$\{t2[0:j-1] = \text{ éléments strictement positifs de } t1[0:n-1]\}$$

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

$$\{n \geq 0, j = 0, i = 0\}$$

```
while (i < n) {
    if (t1[i] > 0) {
        t2[j] = t1[i];
        j++;
    }
    i++;
}
```

$$\{t2[0:j-1] = \text{ éléments strictement positifs de } t1[0:n-1]\}$$

Por 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 :

$$I : 0 \leq j \leq i \leq n$$
$$t2[0:j-1] = \text{ éléments strictement positids de } t1[0:i-1]$$

Cet invariant exprime que $t2$ contiendra dans les j premiers éléments, les i premiers éléments de $t1$ qui sont strictement positifs.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $i = 0$ et $j = 0$. On a bien que tous les éléments strictement positifs du sous-tableau vide sont dans le deuxième sous-tableau qui est lui aussi vide.
- Pour chaque itération de la boucle, on a le triple :

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

```
if (t1[i] > 0) {
    t2[j] = t1[i];
    j++;
}
i++;
```

$$\{I\}$$

Montrons que ce triplet est valide. Notons i et i', j et j', et $t2$ et $t2'$ les valeurs des variables avant et après une itération spécifique.
Pour chaque itération, on a $i' = i + 1$. Nous devons traiter 2 cas :

- Si $t1[i] \leq 0$, alors on a $j' = j$ et $t2' = t2$. Rien n'est ajouté au tableau $t2$ car le nouvel élément n'est pas strictement positif. Comme on avait que $t2[0 : j-1]$ contenait les éléments strictement positifs de $t1[0 : i-1]$, $t2'[o : j'-1]$ contient toujours les éléments strictement positifs de $t1[0 : i'-1]$.
- Sinon, on a $j' = j + 1$ et $t2' = t2[0 : j-1]t1[i]$. On a ajouté à la fin le nouvel élément strictement positif trouvé. On a bien que le sous-tableau $t2'[0 : j'-1]$ contient tous les éléments strictement positifs de $t2[0 : i'-1]$.

- En fin de boucle, on a $(I, i = n)$, on a bien alors que $t2[0 : j - 1]$ contient les éléments strictement positifs de $t1[0 : n - 1]$.

Un variant possible est $v = n - i$.

19. (a) Pour résoudre ce problème, on va parcourir la chaine de caractères. Dès qu'un des caractères n'est pas un chiffre, on va retourner 0.

```
unsigned only_nb(char *s) {
    while (*s) {
        if (*s < '0' || *s > '9')
            return 0;
        s++;
    }

    return 1;
}
```

(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 :

$$\{s_0 = s\}$$

```
while (*s) {
    if (*s < '0' || *s > '9')
        return 0;
    s++;
}
```

$$\{\text{Valeur de retour } \neq 0 \text{ ssi la chaine } s_0 \text{ ne contient que des chiffres}\}$$

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 :

$$\begin{aligned} I :& \exists k \geq 0 : s = s_0 + k \\ & s_0[0 : k - 1] \text{ ne contient que des chiffres} \end{aligned}$$

Cet invariant exprime que tant qu'on est dans la boucle, la sous-chaine avant le caractère actuel ne contient que des chiffres.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $s = s_0$ et donc $k = 0$. On a bien que tous les caractères de la sous-chaine vide, on n'a pas trouvé de caractère qui n'est pas un chiffre.
- Pour chaque itération de la boucle, on a le triplet :

$$\{I, *s\}$$

```
if (*s < '0' || *s > '9')
    return 0;
s++;
```

$\{I\}$

Montrons que ce triplet est valide. Notons s et s' la valeur de la variable avant et après une itération spécifique.
Nous avons deux cas à traiter.

- Si le caractère actuel est un chiffre, alors $s' = s + 1$ et on a bien que tous les caractères entre s_0 et $s' - 1$ (k caractères) sont des chiffres.
- Sinon, $s' = s$ et on quitte la fonction. Tous les caractères entre s_0 et $s' - 1$ sont bien des chiffres.

- En fin de boucle, on a deux cas :
  - $(I, ! * s)$ ce qui signifie que nous avons parcouru toute la chaine et il n'y avait que des chiffres. On a bien que la sous-chaine s_0 jusqu'à $s - 1$ contient que des chiffres et la longueur de cette chaine est de k.
  - $(I, *s$ n'est pas un chiffre$)$ $*s$ n'est pas un chiffre mais tous les caractères de la sous-chaine s_0 $s - 1$ sont des chiffres.

20. (a) Cette fonction parcourt un à un les caractères qui composent la chaine. Chaque fois qu'un caractère est compris entre `'0'` et `'9'` on ajoute 1. La fonction compte donc le nombre de chiffres dans la chaine de caractères.

    (b) Cette fonction va effectuer un nombre d'appels égal à la taille de la chaine de caractères +1. La complexité en temps de cette fonction est donc $\mathcal{O}(n)$, où n correspond à la taille de la chaine de caractères.

    (c) On peut réécrire cette fonction de manière itérative de la façon suivante :

```
unsigned f(char s[]) {
    unsigned n;

    for(n = 0; s[0]; s++) {
        if (s[0] >= '0' && s[0] <= '9')
            n++;
    }

    return n;
}
```

21. (a) Cette fonction va prendre tous les éléments d'indice pair du tableau et va les additionner entre eux.

    (b) Nous allons parcourir tous les éléments pairs du tableau. Nous allons donc faire $\frac{nb}{2} + 1$ appels. Ceci correspond à une complexité en temps $\mathcal{O}(nb)$.

    (c) On peut réécrire cette fonction de manière itérative de la façon suivante :

```
long f(int t[], int nb) {
    long n;

    for (n = 0.0, int i = 0; i < nb; i+=2)
        n += t[i];

    return n;
}
```

22. (a) Cette fonction retourne le produit des valeurs absolues des éléments du tableau.

(b) Nous allons parcourir le tableau élément par élément. Si un élément est négatif, on rappelle la fonction sans avancer dans le tableau. Au pire des cas, nous allons effectuer $2 \times n$ appels. Ceci correspond au cas où tous les éléments sont négatifs. La complexité en temps est $\mathcal{O}(n)$.

(c) On peut réécrire cette fonction de manière itérative de la façon suivante :

```
double f(double t[], unsigned n) {
    double a = t[0];

    for (unsigned i = 1; i < n; i++) {
        if (t[i] < 0.0)
            a *= (-t[i]);
        else
            a *= t[i];
    }

    return a;
}
```

23. (a) On peut décrire cet algorithme en pseudocode de la façon suivante :

```
bubbleSort(t, n)
    i := 0
    while i < n - 1
        j := 0
        while j < n - i - 1
            if t[j] > t[j+1]
                echange(t[i], t[j])
            endIf
            j := j + 1
        endWhile
        i := i + 1
    endWhile
```

(b) On va parcourir le tableau 1 fois pour chaque valeur et à chaque fois, dans le pire des cas, on devra échanger l'élément jusqu'à arriver au bout du tableau. La complexité en temps est donc $\mathcal{O}(n^2)$.

(c) L'algorithme peut s'écrire en C de la manière suivante :

```
void bubbleSort(int t[], unsigned n) {
    for (unsigned i = 0; i < n - 1; i++) {
        for (unsigned j = 0; j < n - i - 1; j++) {
            if (t[j] > t[j+1]){
                int tmp = t[j];
                t[j] = t[j+1];
                t[j+1] = tmp;
            }
        }
    }
}
```

24. (a) On peut décrire cet algorithme en pseudocode de la façon suivante :

```
selectionSort(t, n)
    i := 0
    while i < n - 1
        min := i
        j := i + 1
        while j < n
            if t[j] < t[min]
                min = j
            endIf
            j := j + 1
        endWhile
        echange(t[i], t[min])
        i := i + 1
    endWhile
```

(b) Nous allons itérer sur chaque élément du tableau et à chaque itération, nous allons également itérer sur le reste du tableau. La complexité théorique en temps est donc $\mathcal{O}(n^2)$.

(c) L'algorithme peut s'écrire en C de la manière suivante :

```
voir selectionSort(int t[], unsigned n) {
    for (unsigned i = 0; i < n - 1; i++) {
        unsigned min = i;
        for (unsigned j = i + 1; j < n; j++) {
            if (t[j] < t[min])
                min = j;
        }
        int tmp = t[i];
        t[i] = t[min];
        t[min] = tmp;
    }
}
```

25. Il suffit de reprendre une des deux fonctions, de changer le type de l'argument t en `char * t[]` et de faire une fonction qui compare caractère par caractère une élément du tableau de chaines de caractères à la place de la comparaison entre les 2 éléments entiers du tableau.

26. (a) On peut décrire cet algorithme en pseudocode de la façon suivante :

```
countingSort(t, n)
    max := 0
    i := 0
    while (i < n)
        if (t[i] > max)
            max = t[i]
        endIf
        i := i + 1
    endWhile

    count[max+1] := 0
    i := 0
    while (i < n)
        count[t[i]] := count[t[i]] + 1
        i := i + 1
```

```
endWhile

i := 1
while (i <= max)
    count[i] := count[i] + count[i-1]
    i := i + 1
endWhile

out[n]
i := n - 1
while (i >= 0)
    output[count[t[i]]-1] := t[i];
    count[t[i]] := count[t[i]] - 1
    i := i - 1

i := 0
while (i < n)
    t[i] := output[i]
    i := i + 1
```

(b) La complexité en temps et en espace est identique. Nous avons plusieurs boucles, elles parcourent le tableau t de taille n et le tableau *count* de taille max. La complexité en temps est donc $\mathcal{O}(n + max)$. Pour la complexité en espace, on a un tableau de taille n et on crée un deuxième tableau de taille max. Le complexité est donc de $\mathcal{O}(n + max)$.

(c) On peut juste traduire le pseudocode en C en mettant un tableau de char à la place.

```
void countingSort(char t[], unsigned n) {
    unsigned max = 0;

    for (unsigned i = 0; i < n; i++)
        if(t[i] - 'a' > max)
            max = t[i] - 'a';

    int *count;
    count = calloc(max+1, sizeof(int));

    for (unsigned i = 0; i < n; i++)
        count[t[i]-'a']++;

    for (unsigned i = 1; i <= max; i++)
        count[i] += count[i-1];

    char *output;
    output = malloc(sizeof(char)*n)
    for (int i = n - 1; i >= 0; i--) {
        output[t[i]-'a'-1] = t[i];
        count[t[i]-'a']--;
    }

    for (unsigned i = 0; i < n; i++)
        t[i] = output[i];
```

```
    free(count);
    free(output);

}
```

27. (a) Il suffit d'utiliser notre tri par comptage avec nos chaines de caractères puis de comparer les deux chaines nouvellement triées.

    (b) La complexité théorique en temps par notre algorithme précédent : $\mathcal{O}(n + max)$.

    (c) Voici une fonction C appliquant ce qu'on cherche à faire :

```
unsigned anagrame(char t1[], char t2[], unsigned n) {
    countingSort(t1, n);
    countingSort(t2, n);
    for (unsigned i = 0; i < n; i++)
        if (t1[i] != t2[i])
            return 0;
    return 1;
}
```

28. (a) Voici un pseudocode pour le quicksort :

```
partition(t, petit, grand)
    pivot := t[petit]
    res := grand
    i := grand
    while (i > petit)
        if (t[i] > pivot)
            echange t[i] et t[res]
            res := res - 1
        endIf
        i := i - 1
    endWhile

quickSort(t, petit, grand)
    if (petit < grand)
        indice := partition(t, petit, grand)
        quickSort(t, petit, indice - 1)
        quickSort(t, indice + 1, grand)
    endIf
```

    (b) Le pire des cas arrive quand la fonction de partition retourne la taille du vecteur -1. Ceci arrive si le pivot est le plus grand ou le plus petit élément de la liste. Dans ce cas, on fera au pire $n \times n - 1$ appels, et donc la complexité en temps est de $\mathcal{O}(n^2)$. Nous noterons cependant qu'en moyenne, le quicksort a une complexité en temps $\mathcal{O}(n \log n)$.

    (c) Voici une implémentation du quicksort :

```
int partition(int t, int petit, int grand) {
    int pivot = t[petit];
    int debut = petit;
    int fin = grand;
    int res = grand;
```

```
        for (int i = grand; i > petit; i--)
            if (t[i] > pivot) {
                int tmp = t[i];
                t[i] = t[res];
                t[res] = tmp;
                res--;
            }

        return res;
    }

    void quickSort(int t, int petit, int grand) {
        if (petit < grand) {
            int indice = partition(t, petit, grand);

            quickSort(t, petit, indice - 1);
            quickSort(t, indice + 1, grand);
        }
    }
```

29. On peut réécrire l'algorithme du tri par fusion de manière itérative de la façon suivante :

```
void merge(int t[], int debut, int milieu, int fin) {
    int tailleGauche = milieu - debut + 1;
    int tailleDroite = fin - milieu;
    int *gauche, *droite;
    gauche = malloc(sizeof(int)*tailleGauche);
    droite = malloc(sizeof(int)*tailleDroite);

    for (int i = 0; i < tailleGauche; i++)
        gauche[i] = t[debut + i];
    for (int i = 0; i < tailleDroite; i++)
        droite[i] = t[milieu + i + 1];

    int i = 0, j = 0, k = 1;
    while (i < tailleGauche && j < tailleDroite) {
        if (gauche[i] <= droite[j]) {
            t[k] = gauche[i];
            i++;
        }
        else {
            t[k] = droite[j];
            j++;
        }
        k++;
    }

    while (i < tailleGauche) {
        t[k] = gauche[i];
        i++;
        k++;
    }
```

```
    while (j < tailleDroite) {
        t[k] = droite[j];
        j++
        k++
    }

}

void mergeSort(int t[], int n) {
    for (int taille = 1; taille < n; taille = 2*taille) {
        for (int debut = 0; debut < n - 1; debut += 2*taille) {
            int milieu;
            if (debut + taille - 1 < n-1)
                milieu = debut + taille - 1;
            else
                milieu = n-1;
            int fin;
            if (debut + 2*taille - 1 < n-1)
                fin = debut + 2*taille - 1;
            else
                fin = n-1;
            merge(t, debut, milieu, fin);
        }
    }
}
```

30. (a) On peut réécrire l'algorithme du tri par fusion de cette manière :

```
int sous_vec(int t[], unsigned debut, unsigned fin, int x) {
    if(fin < debut)
        return 0;

    if (debut == fin)
        retunr t[debut] == x;

    int milieu = debut + (fin - debut)/2;

    if (t[milieu] == x)
        return 1;

    if (t[milieu] > x)
        return sous_vec(t, debut, milieu-1, x);
    return sous_vec(t, milieu + 1, fin, x);
}

int recherche(int t[], unsigned n, int x) {
    return sous_vec(t, 0, n-1, x)
}
```

(b) On effectue chaque fois un appel à sous_vec avec une taille totale (entre debut et fin) de la taille actuelle divisée par 2. Le pire des cas revient au cas où on doit parcourir jusqu'à ce que debut $\geq$

fin, ce qui revient à faire $\log_2 n$ appels, où n est la taille du (sous-)vecteur. Comme aucune boucle n'est présente et qu'aucun tableau n'est créé, la complexité théorique en temps et en espace sont identiques et sont $\mathcal{O}(\log n)$, où n correspond à la taille du vecteur.