Correction des exercices supplémentaires du chapitre 4

1. (a) Pour résoudre ce problème, on peut juste compter le nombre de fois que n est divisible par f. Nous pouvons réaliser ceci grâce à l'opération modulo et en divisant chaque fois n par f.

```
unsigned f(unsigned n, unsigned f){
    unsigned m;

    for (m = 0; (n%f) == 0; m++, n/=f);

    return m;
}
```

   (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 :

$$\{n = n_0 > 0, f > 0\}$$

```
for (m = 0; (n%f) == 0; m++, n\=f);
```

$$\{m = \text{ multiplicité de f dans } n_0\}$$

   Où n_0 dénote la valeur initiale de n. En décomposant la boucle `for` en boucle `while`, on obtient :

$$\{n = n_0 > 0, f > 0, m = 0\}$$

```
while ((n%f) == 0){
    m++;
    n \= f;
}
```

$$\{m = \text{ multiplicité de f dans } n_0\}$$

   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 :

$$\begin{aligned} I :& n > 0 \\ & f > 0 \\ & n = \frac{n_0}{f^m} \end{aligned}$$

   Cet invariant exprime qu'au début et à la fin de chaque itération, la valeur actuelle de n correspond à la valeur initiale de n divisée par f à la puissance m.

   Montrons maintenant que cet invariant est valide.

   - Initialement, on a $n = n_0 > 0$, $f > 0$ et $m = 0$ On a bien que $n = \frac{n_0}{f^0} = \frac{n_0}{1}$.
   - Pour chaque itération de boucle, on a le triplet :

$$\{I, n \text{ est divisible par } f\}$$

```
m++;
n \= f;
```

$\{I\}$

Montrons que ce triplet est valide. Notons n, n' et m m' les valeurs de n et m avant et après une certaine itération.
À chaque itération, nous obtenons que :

- $m' = m + 1$
- $n' = \frac{n}{f}$
- et donc $\frac{n}{f} = \frac{n_0}{f^{m+1}} \Leftrightarrow n' = \frac{n_0}{f^{m'}}$

- En fin de boucle, on a $\{I, \not((n\%f) == 0)\}$, donc n n'est plus divisible par f et on a bien alors que $n = \frac{n_0}{f^m} \Leftrightarrow n_0 = n \times f^m$, m est bien la multiplicité de f dans n_0.

2. (a) Pour résoudre ce problème, nous pouvons juste énumérer tous les diviseurs de n entre 1 et $\sqrt{n}$ Pour chaque diviseur, on doit compter celui-ci ainsi que le résultat de la division qui divise également d. Nous avons juste un cas particulier où $d = \frac{n}{d}$.

```
unsigned nb_div(unsigned n) {
    unsigned nb = 0;

    for (unsigned d = 1; d * d <= n; d++) {
        if (!(n%d)) {
            if (d == n/d)
                nb++;
            else
                nb+=2;
        }
    }
    return nb;
}
```

   (b) Pour une valeur donnée de n, le nombre d'itérations effectuées de la boucle sera bornée par $\sqrt{n}$. La complexité en temps de la fonction vaut donc $\mathcal{O}(\sqrt{n})$. En ce qui concerne la complexitée en espace, aucun tableau n'est utilisé et comme la fonction n'est pas récursive, rien n'est mis sur la pile d'exécutions. Nous avons donc une complexité $\mathcal{O}(1)$.

   (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 :

$\{n > 0, nb = 0\}$

```
for (unsigned d = 1; d * d <= n; d++) {
    if (!(n%d)) {
        if (d == n/d)
            nb++;
        else
            nb+=2;
    }
}
```

$$\{nb = \text{ nombre de diviseurs de } n\}$$

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

$$\{n > 0, nb = 0, d = 1\}$$

```
while (d * d <= n) {
    if (!(n%d)) {
        if (d == n/d)
            nb++;
        else
            nb+=2;
    }
    d++;
}
```

$$\{nb = \text{ nombre de diviseurs de } n\}$$

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 :

$$\begin{aligned} I :& n > 0 \\ & d \leq \sqrt{n} + 1 \\ & nb = \text{ nombre de diviseurs } k \text{ de } n \text{ tels que } k < d \text{ ou } k > \frac{n}{d} \end{aligned}$$

Cet invariant exprime qu'au début et à la fin de chaque itération, la valeur de nb correspond au nombre de diviseurs de n qui sont compris entre $[1, d[$ et entre $]\frac{n}{d}, n]$.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $n > 0, d = 1, nb = 0$. En effet, il n'y a aucun diviseur de n dans le sous-ensemble vide de potentiels diviseurs.
- Pour chaque itération de la boucle, on a le triplet :

$$\{I, d \leq \sqrt{n}\}$$

```
if (!(n%d)) {
    if (d == n/d)
        nb++;
    else
        nb+=2;
}
d++;
```

$$\{I\}$$

Montrons que ce triplet est vialide. Notons d, d' et nb, nb' les valeurs de n et nb avant et après une itération spécifique.
À chaque itération, nous obtenons que $d' = d + 1$.
Nous avons 3 cas à traiter :

i. n n'est pas divisible par d : dans ce cas, d n'est pas un diviseur et on ne doit pas compter d'autre diviseur. On a bien que $nb' = nb$

ii. n est divisible par d et $\frac{n}{d} = d$. Dans ce cas, $nb' = nb + 1$. Comme d et $\frac{n}{d}$ sont identiques, il n'y a qu'un nouveau diviseur à compter. On a bien $nb + 1 = nb'$ entre $[1, d+1[= [1, d'[$ et $]\frac{n}{d+1}, n] =]\frac{n}{d'}, n]$.

iii. n est divisible par d et $\frac{n}{d} \neq d$. Dans ce cas, $nb' = nb + 2$. Comme d et $\frac{n}{d}$ divisent tous les deux n, il y a deux nouveaux diviseurs à compter. On a bien $nb + 2 = nb'$ entre $[1, d+1[= [1, d'[$ et $]\frac{n}{d+1}, n] =]\frac{n}{d'}, n]$.

- En fin de boucle, on a $(I, d * d > n)$. Comme $d > \sqrt{n}$, on a bien vérifié tous les potentiels diviseurs k tels que $k < d$ ou $k > \frac{n}{d}$.
- Il reste à démontrer que la boucle se termine toujours. On peut considérer le variant

$$v = \lfloor \sqrt{n} \rfloor - d + 1$$

3. (a) Nous pouvons utiliser le même principe que l'exercice précédent. Dès que nous trouvons un diviseur m, on sait que $\frac{n}{m}$ est le plus grand diviseur $< n$. Si le nombre est premier, alors le plus grand diviseur est 1.

```
unsigned big_div(unsigned n) {
    unsigned m;

    for (m = 2; m * m <= n; m++)
        if (!(n%m))
            return n/m;
    return 1;
}
```

   (b) Le nombre d'itérations effectuées est borné par $\sqrt{n}$ donc la complexité en temps est $\mathcal{O}(\sqrt{n})$. Pour la complexité en espace, aucun tableau n'est utilisé et comme la fonction n'est pas récursive, rien n'est mis sur la pile d'exécutions. Nous avons donc une complexité $\mathcal{O}(1)$.

4. (a) Afin de récupérer tous les chiffres d'un nombre, il suffit de chaque fois récupérer le reste de sa division entière par 10. Ceci est possible grâce à l'opérateur modulo.

```
unsigned sum_chif(unsigned n) {
    unsigned sum = 0;

    while (n) {
        sum += (n%10);
        n \= 10;
    }
    return sum;
}
```

   (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 :

$\{sum = 0, n = n_0\}$

```
while (n) {
    sum += (n%10);
    n /= 10;
}
```

$$\{sum = \text{ somme des chiffres de } n_0\}$$

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 :$n_0 \geq 0$

$\forall i$ correspondant à l'itération actuelle, $\exists r \in \mathcal{N} : n_0 = 10^i \times n + r$ et $0 \leq r \leq 10^i$

$sum =$ somme des chiffres de r

Cet invariant exprime qu'avant et après chaque itération de la boucle, la valeur de n correspond au quotient entier de la valeur initiale n_0 de cette variable par une certaine puissance de 1. Cela revient à dire que l'écriture décimale de n est un préfixe de celle de n_0 .

Montrons maintenant que cet invariant est valide.

- Initialement, on a $n = n_0$ et $sum = 0$. On a bien que $n_0 = 10^0 \times n_0 + 0$ et somme des chiffres de 0 est 0.
- Pour chaque itération de la boucle, on a le triplet :

$$\{I, n \neq 0\}$$

```
sum += (n%10);
n /=10;
```

$$\{I\}$$

Montrons que ce triplet est valide. Notons n, n' et sum, sum' les valeurs avant et après une certaine itération. De plus i et i' correspondent au nombre d'itérations effectuées avant et après chaque itération.
À chaque itération, nous obtenons que

- $sum' = sum + d$, où d est le reste de la division de n par 10.
- $n' = \frac{n-d}{10}$
- On a bien que $n_0 = 10^{i+1} \times \frac{n-d}{10} + r = 10^{i'} \times n' + r'$ où $r' = 10^i d + r$.En plus, on a toujours $0 \leq r < 10^{i'}$.

- En fin de boucle, on a $\{I, n = 0\}$ ce qui implique que la valeur de sum est égale à la somme des chiffres de n_0. En effet, si $n = 0$, alors $n_0 = 10^i \times n + r \Rightarrow r = n_0$

5. (a) Pour savoir si un nombre donné se termine par 0 dans son écriture décimale, il suffit de vérifier que son modulo 10 vaut 0.

```
unsigned nb_zero(unsigned n) {
    unsigned nb = 0;

    while (!(n%10)){
        nb++;
        n/=10;
    }

    return nb;
}
```

(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 :

$$\{nb = 0, n = n_0\}$$

```
while (!(n%10)){
    nb++;
    n/=10;
}
```

$$\{nb = \text{ nombre de zéros à la fin de l'écriture de } n_0\}$$

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 `whilw`, et doit impliquer la postcondition après la dernière itération. Un invariant possible est :

$$\begin{aligned} I : & n_0 > 0 \\ & nb \geq 0 \\ & n_0 = n \times 10^{nb} \end{aligned}$$

Cet invariant exprime qu'avant et après chaque itération, la valeur de `nb` correspond au nombre de facteurs 10 qui ont été extraits de la valeur initiale de n.

Montrons maintenant que cet invariant est valide.

- Initialement, on a $n = n_0$ et $nb = 0$. On a bien que $n_0 = n \times 10^0$.
- Pour chaque itération de la boucle, on a le triplet :

$$\{I, n \text{ est divisible par } 10\}$$

```
nb++;
n /=10;
```

$$\{I\}$$

Montrons que ce triplet est valide. Notons n, n' et nb, nb' les valeurs avant et après une certaine itération.
À chaque itération, nous obtenons que
  - $n' = n/10$
  - $nb' = nb$
  - Nous obtenons alors que $n_0 = \frac{n}{10} \times 10^{nb+1} = n' \times 10^{nb'}$
- En fin de boucle, on a $\{I, n \text{ n'est pas divisible par } 10\}$. On a donc $n_0 = n \times 10^{nb}$ avec n non-divisible par 10. Cela signifie que nb est la plus grande puissance de 10 qui divise n_0, en d'autres termes, que nb est égal au nombre de zéros situés à la fin de l'écriture de n_0.

6. (a) le pgcd de 1 avec un autre nombre est toujours 1, nous pouvons donc directement commencer l'énumération à partir de 2.

```
unsigned euler(unsigned n) {
    unsigned e = 1;

    for (unsigned m = 2; m <= n; m++)
        if ( pgcd(m, n) == 1)
            e++;

    return e;
}
```

(b) Nous allons itérer un nombre $n-1$ fois dans la boucle. Cette boucle contient une fonction qui a une complexité théorique $\mathcal{O}(\log n)$. Nous allons avoir une complexité en temps de $n \times \log n = \mathcal{O}(n \log n)$.

7. (a) Le plus petit diviseur différent de 1 d'un nombre est inférieur ou égal à $\sqrt{n}$, ou le nombre lui-même s'il s'agit d'un nombre premier.

```
unsigned petit_div(unsigned n) {
    for (unsigned d = 2; d * d <= n; d++)
        if (!(n%d))
            return d;

    return n;
}
```

(b) On itère au plus $\sqrt{n}$ fois. Chaque opération se fait en temps constant. La fonction a donc une complexité en temps théorique $\mathcal{O}(\sqrt{n})$.

8. (a) Pour résoudre ce problème, nous pouvons tout simplement énumérer tous les diviseurs m de n jusqu'à $\sqrt{n}$ et regarder si m est premier ainsi que $\frac{m}{n}$.

```
unsigned nfp(unsigned n) {
    unsigned m = 0;

    for (unsigned d = 1; d * d <= n; d++) {
        if (!(n%d)){
            if (premier(d))
                m ++;
            if (premier(n/d) && d != n/d)
                m++;
        }
    }

    return m;
}
```

(b) Nous allons itérer n fois dans la boucle. Comme la fonction premier a une complexité constante en temps ainsi que toutes les autres opérations, nous avons une complexité théorique en temps qui est $\mathcal{O}(\sqrt{n})$

9. (a) Cette fonction retourne le nombre de chiffres valant 9 d'un nombre écrit en base 10, représenté par la variable v.

(b) Cette fonction va à chaque fois s'appeler avec l'argument divisé par 10. Un appel à la fonction avec un argument $v > 0$ va créer $\lfloor \log_{10} v \rfloor + 1$ appels à la fonction. La complexité théorique est donc $\mathcal{O}(\log v)$.

(c) On peut réécrire cette fonction de cette manière :

```
unsigned f(unsigned v) {
    unsigned n = 0;

    while (v) {
        if ((v%10) == 9)
            n++;
        v/=10;
    }

    return n;
}
```

10. (a) L'appel de `f` avec $n\%2$ se fera toujours en temps constant. L'évaluation de `f(n/2)` appelle récursivement la fonction avec un argument divisé par 2 à chaque appel avant d'atteindre le cas de base quand il est plus petit ou égal à 1. Le nombre d'appels est de $\log_2 n$. La complexité en temps et en espace est identique et est $\mathcal{O}(\log n)$.

(b) On peut réécrire la fonction comme :

```
unsigned f(unsigned n) {
    unsigned m;

    for (m = 0; n; n/=2)
        m += (n%2);

    return m;
}
```

11. (a) Cette fonction calcule x^{2^n}.

(b) Cette fonction peut se réécrire :

```
double f(double x, unsigned n) {
    for (unsigned i = 0; i < n; i++)
        x *= x;

    return x;
}
```

12. (a) Cette fonction retourne l'exposant de la plus grande puissance de 2 qui divise n.

(b) La complexité en espace de cette fonction récursive correspond à sa profondeur de récursion ou encore le nombre d'exécutions mis sur la pile d'exécutions. Nous divisons chaque fois l'argument par 2 et le pire cas arrive quand nous avons un nombre n qui est une puissance de 2. La complexité en espace est $\log_2 n = \mathcal{O}(\log n)$

(c) On peut réécrire cette fonction :

```
unsigned f(unsigned long n) {
    unsigned r;

    for (r = 0; n && !(n%2); n/=2, r++);

    return r;
}
```