Correction des exercices sur la récursivité

1. Une façon simple de résoudre le problème consiste à diviser répétitivement n par f tant que cela est possible, et à compter le nombre de fois où cette opération est effectuée. On obtient le code suivant

```
unsigned multiplicite_facteur(unsigned n, unsigned f){
    unsigned m;
    for (m = 0; !(n % f); m++)
        n /= f;
    return m;
}
```

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

```
for (m = -; !(n % f); m++)
    n /= f;
```

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

où n_0 dénote la valeur intiale de n. En développant la boucle *for*, cela équivaut à démontrer

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

```
while (!(n % f)){
    n /= f;
    m++;
}
```

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

$$I : n > 0 \& f > 0 \& n_0 = n.f^m.$$

- Initialement, on a $(n = n_0 > 0 \& f > 0 \& m = 0)$
- D'une itération à l'autre de la boucle, on a le triplet

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

```
n /= f;
n++;
```

$$\{I\}$$

qui est bien valide. En effet, si n' et m' dénotent respectivement les valeurs de n et m après l'exécution du fragment de code, on

$$n' = \frac{n}{f} \& m' = m + 1$$

donc I et n est divisible par f, ce qui implique $n' > 0 \& f > 0 \& n_0 = n.f^m = n'.f^{m'}$

- En sortie de boucle, on a I et n n'est pas divisible par f qui implique bien $m =$ multiplixité de f dans n_0, car si $n_0 = n.f^m$ et n n'est pas divisible par f, alors m est la multiplictié de f dans n_0.

2. Il suffit d'énumérer tous les diviseurs potentiels d de n dans l'intervalle $[1, \sqrt{n}]$. Pour chaque valeur de d qui divise n, on compte les deux diviseurs d et n/d, ou un seul dans le cas particulier où ces deux valeurs sont égales. On obtient le code suivant

```
unsigned nb_diviseurs(unsigned n){
    unsigned nb, d;
    for (d = 1, nb = 0; d * d <= n; d++){
        if (n % d)
            continue;
        if (d == n / d)
            nb++;
        else
            nb += 2;
    }
    return nb;
}
```

Pour une valeur donnée de n, le nombre d'itérations de la boucle est borné par $\sqrt{n}$. La complexité en temps de la fonction vaut donc $\mathcal{O}(\sqrt{n})$. La quantité de mémoire consommée par la fonction est une constante indépendante de n. Sa complexité en espace vaut donc $\mathcal{O}(1)$.

$\{n > 0\}$

```
for (d = 1, nb = 0; d * d <= n; d++){
    if (n % d)
        continue;
    if (d == n / d)
        nb++;
    else
        nb += 2l
}
```

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

Ce triplet est équivalent à

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

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

```
            else
                nb += 2;
        }
        d++;
    }
```

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

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

Montrons que cet invariant est valide.

- Initialement, on a $\{n > 0, d = 1, nb = 0\} \implies I$
- 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 valide, en notant respectivement x et x' la valeur d'une variable x avant et après l'itération concernée.

  - Si d ne divise pas n, alors on a $nb' = nb$ et $d' = d+1$. Ni d, ni n/d ne sont des diviseurs de n, donc la postcondition est satisfaite.
  - Si d divise n et $d = n/d$, alors on a $nb' = nb + 1$ et $d' = d + 1$. Le nombre $d = n/d$ est un diviseur de n tel que $d < d'$, donc la postcondition est satisfaite.
  - Si d divise n et $d \neq n/d$, alors on a $nb' = nb + 2$ et $d' = d + 1$. Les nombres d et n/d sont deux diviseurs distincts de n tels que $d < d'$ et $n/d > n/d'$, donc la postcondition est satisfaite.

- En fin de boucle, on a $\{I, d > \sqrt{n}\} \implies \{nb =$ nombre de diviseurs de $n\}$. En effet, si $d > \sqrt{n}$, alors tous les diviseurs θ de n satisfont l'une des conditions $\theta < d$ ou $\theta > n/d$.

Il reste à montrer que la boucle se termine toujours. Il suffit de considérer le variant

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

où $\lfloor x \rfloor$ dénote le plus grand entier inférieur ou égal à x. Cette expression possède en effet une valeur entière non-négative, qui décroit à chaque itération de la boucle. La terminaison de cette boucle est donc toujours garantie.

3. Pour énumérer les diviseurs de n, on peut utiliser la même stratégie que celle vue au cours pour la recherche de nombres parfaits : il suffit de considérer tous les entiers contenus dans l'intervalle $[2, \sqrt{n}]$, et de trouver le plus petit d'entre eux qui divise n. Si ce nombre est noté i, alors le diviseur recherché vaut $d = n/i$. Si aucun diviseur n'est trouvé, alors on a $d = 1$. On obtient alors le code suivant.

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

Le nombre d'itérations effectuées est borné par $\sqrt{n}$, donc la complexité en temps de cette fonction est $\mathcal{O}(\sqrt{n})$. La fonction consomme un espace mémoire de taille constante, donc sa complexité en espace est $\mathcal{O}(1)$.

4. L'évaluation de $f(n\%2)$ appelle la fonction f avec un argument égal à 0 ou 1. Cet appel s'exécute en temps constant et retourne une valeur égale au reste de la division de n par 2. L'évaluation de $f(n/2)$ appelle récursivement la fonction f, en divisant l'argument par 2 à chaque appel jusqu'à atteindre une valeur inférieure ou égale à 1. La profondeur de récursion et le nombre d'appels récursifs sont donc tous les deux $\mathcal{O}(\log n)$. En résumé, on a donc des complexités en temps et en espace qui sont toutes les deux $\mathcal{O}(\log n)$.

```
unsigned f(unsigned n){
    unsigned r;
    for (r = 0; n; n /= 2)
        r += n % 2;
    return r;
}
```

5. Cette fonction compte le nombre de chiffres égaux à 9 dans l'écriture décimale du nombre v qui lui est passé en argument.
Pour un nombre $v > 0$ donné, l'évaluation de $f(v)$ provoque $\lfloor \log_{10} v \rfloor + 1$ appels à la fonction f, où $\lfloor x \rfloor$ dénote le plus grand entier inférieur ou égal à x. Chacun de ces appels exécute un nombre borné d'instructions. La complexité en temps de cette fonction vaut donc $\mathcal{O}(\lfloor \log_{10} v \rfloor + 1) = \mathcal{O}(\log v)$.

```
unsigned f(unsigned v){
    int n;
    for (n = 0; v; v /= 10)
        if (v % 10 == 9)
            n++;
    return n;
}
```

6. .

```
unsigned euler(unsigned n){
        unsigned e, m;
        for (e = 1, m = 2; m <= n; m++)
                if (pgcd(m, n) == 1)
                        e++;
        return e;
}
```

Le nombre total d'opérations effectuées est borné par :

$$\begin{aligned}\mathcal{O}(\log 2 + \log 3 + \ldots + \log n) &\leq \mathcal{O}(\log n + \log n + \ldots + \log n) \\ &= \mathcal{O}((n-1)\log n) \\ &= \mathcal{O}(n \log n)\end{aligned}$$