TD #1 — Flot de contrôle, tests, analyse de coût (tout ça ?!) (avec solutions)

Exercice 1. *Affichage joli*

On se donne le programme C suivant :

```c
void pretty_print(uint32_t x) {
    if (x > 1000000000) {
        printf("%u ", x / 1000000000);
        x = x % 1000000000;
    }
    if (x > 1000000) {
        printf("%u ", x / 1000000);
        x = x % 1000000;
    }
    if (x > 1000) {
        printf("%u ", x / 1000);
        x = x % 1000;
    }
    printf("%u\n", x);
}

int main(void) {
    uint32_t a = 1;
    int i = 0;
    while (i < 9) {
        a = 2 * a;
        i = i + 1;
    }
    pretty_print(a);

    return EXIT_SUCCESS;
}
```

1. Quels fichiers d'en-tête faudrait il lui ajouter pour qu'il compile ?

   ```c
   #include <stdlib.h>
   #include <stdio.h>
   #include <stdint.h>
   ```

2. Exécutez ce programme « à la main », et déterminez en chaque point d'exécution les valeurs des différentes variables *qui y sont visibles*, ainsi que ce qu'il affiche éventuellement sur la sortie standard.
3. Quelle autre construction aurait on pu utiliser pour implémenter la boucle `while` de la fonction `main` ?

   Une boucle `for`.

4. Réécrivez cette boucle de cette façon.

   Par exemple :

   ```c
   for (int i = 0; i < 9; i++)
   {
       a = 2 * a;
   }
   ```

5. Comment pourrait on modifier cette boucle afin d'exécuter *toutes* les lignes de la fonction *pretty_print* ?

On peut par exemple changer la condition `i < 9` en `i < 30`.

Le spécifieur de conversion %u n'affiche pas les chiffres nuls, ce qui fait que la fonction *pretty_print* proposée n'est en fait pas correcte (pour son usage prévu). Il existe cependant une variante « `%03u` » qui permet d'afficher toujours au moins trois chiffres en insérant si besoin des zéros « à gauche ».

6. Proposez une version corrigée de *pretty_print*. Prenez garde à bien traiter tous les cas.

On propose :

```
void pretty_print(uint32_t x)
{
    uint32_t xo = x;
    if (x > 1000000000)
    {
        printf("%u ", x / 1000000000);
        x = x % 1000000000;
    }
    if (x > 1000000)
    {
        if (x < xo)
        {
            printf("%03u ", x / 1000000);
        }
        else
        {
            printf("%u ", x / 1000000);
        }
        x = x % 1000000;
    }
    if (x > 1000)
    {
        if (x < xo)
        {
            printf("%03u ", x / 1000);
        }
        else
        {
            printf("%u ", x / 1000);
        }
        x = x % 1000;
    }
    if (x < xo)
    {
        printf("%03u\n", x);
    }
    else
    {
        printf("%u\n", x);
    }

    return;
}
```

7. Le test effectué précédemment est-il toujours exhaustif ? Si non, proposez une fonction de test qui permette d'exécuter toutes les lignes de la nouvelle fonction *pretty_print*.

Non, un appel de la fonction avec une unique valeur ne va toujours exécuter qu'au plus une des alternatives des conditions « `x < xo` ». On propose :

```
void test_pp(void)
{
```

```
    uint32_t a = 1;
    int i = 0;
    pretty_print(a);
    for (; i < 10; i++)
    {
        a = 2 * a;
    }
    pretty_print(a);
    for (; i < 20; i++)
    {
        a = 2 * a;
    }
    pretty_print(a);
    for (; i < 30; i++)
    {
        a = 2 * a;
    }
    pretty_print(a);
}
```

Exercice 2. *Kworème*

On se donne la fonction C suivante :

```
void quorem(uint32_t x, uint32_t d, uint32_t out[2]) {
    uint32_t q = 0;
    uint32_t r = x;

    while (r >= d) {
        r = r - d;
        q = q + 1;
    }

    out[0] = q;
    out[1] = r;

    return;
}
```

1. Que semble-t'elle calculer ?

   Cette fonction calcule le quotient et le reste de la division « euclidienne » de x par d, et les écrits dans *out*.

2. Ajoutez-y des assertions « `assert` » afin de vérifier à l'exécution que ses entrées sont (a priori) valides, et que le résultat produit est celui (a priori) attendu.

   On propose :

```
void quorem(uint32_t x, uint32_t d, uint32_t out[2]) {
    uint32_t q = 0;
    uint32_t r = x;

    assert(d > 0);

    while (r >= d) {
        r = r - d;
        q = q + 1;
    }

    assert(r < d);
    assert(x == q * d + r);
```

```
    out[0] = q;
    out[1] = r;

    return;
}
```

3. Cette fonction vous semble-t elle efficace ?

Si l'on double x, il faut 1 bit de plus pour écrire la nouvelle valeur et le temps d'exécution sera multiplié par deux. Le temps d'exécution est donc exponentiel en le nombre de bits nécessaires pour écrire l'argument x, ce qui ne semble pas très efficace.

4. Avez-vous une idée de modification la rendant (beaucoup) plus efficace ?

Plutôt que d'incrémenter q de *1* à chaque itération de la boucle, on peut par une autre boucle (interne) calculer la plus grande puissance de deux e telle que pour la valeur courante de r on a $d2^e \leq r$; on incrémente alors q de 2^e et décrémente r de $d2^e$. Ces valeurs de e peuvent se calculer à chaque fois avec des multiplications par 2 répétées, et donc au prix d'au plus $O(\log r)$ (qui est aussi un $O(\log x)$) opérations arithmétiques . Cet algorithme modifié revient essentiellement à calculer les bits du quotient un par un (en commençant par le bit de poids fort) ; le nombre d'itérations de la boucle externe est alors borné par la taille (en bits) du quotient, soit $\lceil \log q \rceil$, et le nombre total d'opérations arithmétiques effectuées est un $O(\log(q) \times \log(x)) = O(\log^2 x)$. On est donc passé d'un coût exponentiel en la taille $\log x$ de x à un coût *polynomial*.
On propose l'implémentation suivante de cette fonction modifiée :

```
void quorem_fast(uint32_t x, uint32_t d, uint32_t out[2])
{
    uint32_t q = 0;
    uint32_t r = x;

    while (r >= d)
    {
        uint32_t qinc = 2;
        while (qinc * d <= r)
        {
            qinc = qinc * 2;
        }
        qinc = qinc / 2;
        r = r - qinc * d;
        q = q + qinc;
    }

    out[0] = q;
    out[1] = r;

    return;
}
```

5. Supposez que l'on arrive à prouver que la version ci-dessus est correcte (pour les spécifications imaginées). Proposez une démarche permettant de tester la correction de votre variante plus efficace.

On peut vérifier sur un grand nombre d'entrées x, d valides que les deux fonctions donnent les mêmes résultats. Si la version originale est prouvée correcte, tout désaccord impliquerait que la version plus efficace n'est pas correcte. Idéalement, les entrées choisies doivent être les plus variées possibles afin d'augmenter la chance de détecter une éventuelle erreur ; par défaut, on peut les générer aléatoirement.