Devoir surveillé d'informatique

Consignes

- Les programmes demandés doivent être écrits en C, on suppose que les librairies standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.h>`, `<stdassert.h>`, ...) sont déjà importées.
- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

❑ Exercice 1 : *Questions de cours*

On considère l'algorithme suivant :

Algorithme : Multiplier sans utiliser `*`

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$

Sorties : nm

```
r ← 0
tant que m > 0 faire
    m ← m − 1
    r ← r + n
fin
return r
```

1. Donner les valeurs successives prises par les variables m, n et r si on fait fonctionner cet algorithme avec $n = 7$ et $m = 4$. On pourra recopier et compléter le tableau suivant :

	n	m	r
valeurs initiales	7	4	0
après un tour de boucle	7	3	1
après deux tours de boucle	7	2	14
après trois tours de boucle	7	1	21
après quatre tours de boucle	7	0	28

2. Donner une implémentation de cet algorithme en langage C sous la forme d'une fonction `multiplie` de signature `int multiplie(int n, int m)`. On précisera soigneusement la spécification de cette fonction en commentaire dans le code et on vérifiera les préconditions à l'aide d'instructions `assert`.

```c
// Prends en entrée deux entiers positifs n et m et renvoie leur produit nm
int multiplie(int n, int m)
{
    assert(n >= 0 && m >= 0);
    int r = 0;
    while (m > 0)
    {
        m = m - 1;
        r = r + n;
    }
    return r;
}
```

3. Donner la définition d'un variant de boucle, puis prouver que cet algorithme termine.

Un *variant de boucle* est une quantité qui dépend des variables du programmes et est :

1. entière,
2. positive,
3. strictement décroissante.

. Dans l'algorithme ci-dessus, la quantité m est un variant de boucle, en effet :

1. $m \in \mathbb{N}$ par précondition.
2. $m \in \mathbb{N}$ par précondition puis m reste positif car par condition d'entrée dans la boucle $m \geqslant 1$ et dans la boucle on décrémente m donc après un passage dans la boucle m reste positif ou nul.
3. m décroit strictement car m est diminué de 1 lors de chaque passage dans la boucle.

L'algorithme termine car on a trouvé un variant de boucle.

4. Donner la définition d'un invariant de boucle, puis prouver que cet algorithme est correct.

Un *invariant de boucle* est une propriété qui dépend des variables du programme et qui est :

1. vraie avant d'entrer dans la boucle (initiatilisation)
2. reste vraie après un tour de boucle si elle l'était au tour précédent (conservation)

En sortie de boucle, la validité d'un invariant permet de prouver la correction de l'algorithme.
On note, m_0 la valeur initiale de m, montrons que la propriété I : « $r = (m - m_0)n$ » est un invariant de boucle.

1. Avant d'entrée dans la boucle $m = m_0$ donc $(m - m_0)n = 0$ et comme r est initialisé à 0 la propriété I est vérifiée.
2. On suppose I vérifié à l'entrée de la boucle et on note r' (resp. m') les valeurs prises par r (resp. m) au tour de boucle suivant, alors :
   $(m' - m_0)n = (m + 1 - m_0)n$, or I étant vérifié à l'entrée de boucle $(m - m_0)n = r$ donc
   $(m' - m_0)n = r + n$ et comme $r' = r + n$
   $(m' - m_0)n = r'$ et donc I est vérifiée.

En sortie de boucle, puisque $m = 0$, cette invariant prouve que $r = m_0 n$ et donc l'algorithme est correcte.

❑ Exercice 2 : *Puissance*

1. Ecrire une fonction `valeur_absolue` qui prend en argument un entier n et renvoie sa valeur absolue $|n|$.
   On rappelle que : $|n| = \begin{cases} -n & \text{si } n < 0 \\ n & \text{sinon} \end{cases}$

```c
// Renvoie la valeur absolue de n
int valeur_absolue(int n)
{
    if (n < 0)
        return -n;
    return n;
}
```

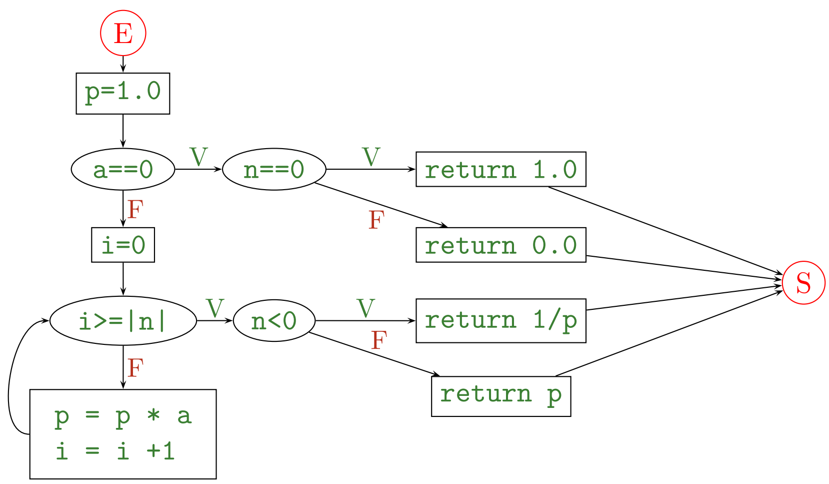
2. Ecrire une fonction `puissance` qui prend en argument un flottant (type `double`) a et un entier n et renvoie a^n. On rappelle que pour $a \in \mathbb{R}^*$, $n \in \mathbb{Z}$:

$$\begin{cases} a^n = \underbrace{a \times \cdots \times a}_{n \text{ facteurs}} & \text{si } n > 0, \\ a^0 = 1, & \\ a^n = \dfrac{1}{a^{-n}} & \text{si } n < 0. \end{cases}$$

D'autre part $0^0 = 1$, $0^n = 0$ si $n \geqslant 0$ et les puissances négatives de zéro ne sont pas définies. On vérifiera la précondition $n \geqslant 0$ lorsque $a = 0$ à l'aide d'une instruction `assert`.

```c
// Renvoie a puissance n (n>=0 si a=0)
double puissance(double a, int n)
{
    double p = 1.0;
    if (a == 0)
    {
        assert(n >= 0);
    }
    if (a == 0)
    {
        if (n == 0)
        {
            return 1.0;
        }
        else
        {
            return 0.0;
        }
    }
    for (int i = 0; i < valeur_absolue(n); i++)
    {
        p = p * a;
    }
    if (n < 0)
    {
        return 1 / p;
    }
    else
    {
        return p;
    }
}
```

3. Tracer le graphe de flot de contrôle de cette fonction.



4. Proposer un jeu de test permettant de couvrir tous les arcs.

Le jeu de test suivant permet de couvrir tous les arcs :

- $a = 0$ et $n = 0$
- $a = 0$ et $n = 1$
- $a = 2$ et $n = 3$
- $a = 2$ et $n = -3$

❑ **Exercice 3** : *Lecture et compréhension d'un code C*
On considère la fonction `mystere` suivante :

```c
1  int mystere(int n)
2  {
3      assert(n > 1);
4      int d = 2;
5      while (n % d != 0)
6      {
7          d = d + 1;
8      }
9      return d;
10 }
```

1. Quelle est la valeurs renvoyée par l'appel `mystere(35)` ? et par `mystere(13)` ?

   `mystere(35)` renvoie 5 et `mystere(13)` renvoie 13.

2. Quel sera le résultat de l'exécution d'un programme effectuant l'appel `mystere(1)` ?

   Le programme s'arrête sur une erreur d'assertion à la ligne 3.

3. Proposer une spécification aussi précise que possible pour cette fonction.

   On peut propose la spécification suivante : « *Prend en entrée un entier n>1 et renvoie son premier diviseur strictement plus grand que 1* ».

4. Prouver la terminaison de cette fonction.

   Montrons que $n - d$ est un variant de boucle :
   — $n - d \in \mathbb{N}$ car n et d sont des entiers.
   — $n - d \geqslant 0$, en effet cela est vrai à l'initialisation ($d = 2$ et $n > 1$) et reste vrai à chaque passage dans la boucle car comme n divise n, par condition d'entrée dans la boucle $d < n$.
   — $n - d$ est strictement croissante car d est incrémenté à chaque tour de boucle.

   Donc cet algorithme termine.

5. En utilisant la fonction précédente, écrire une fonction `est_premier` de prototype :
   `bool est_premier(int n)` qui prend en entrée un entier $n \in \mathbb{N}$ et qui renvoie `true` si et seulement si `n` est premier.

```c
// Renvoie true ssi n est premier
bool est_premier(int n)
{
    assert(n >= 0);
    if (n == 0 || n== 1)
    {
        return false;
    }
    return (mystere(n) == n);
}
```

❑ **Exercice 4** : *Programmation en C : algorithme de Luhn*
Un algorithme (appelé algorithme de Luhn, d'après le nom de son inventeur), est utilisé pour vérifier qu'un numéro de carte de crédit est valide, cela permet d'indiquer à un utilisateur une éventuelle erreur de saisie. Le but de l'exercice est de programmer cet algorithme en C, on prendra soin de préciser dans le code sous forme de commentaire les spécifications des fonctions demandées.

1. Ecrire une fonction `mult2` qui prend en entrée un entier naturel c compris entre 0 (inclus) et 9 (inclus), et renvoie $2c$ si $0 \leqslant 2c \leqslant 9$ et la somme des deux chiffres de $2c$ sinon. Par exemples :
   - `mult2(3)` renvoie 6 (car $2 \times 3 = 6$),
   - `mult2(7)` renvoie 5 (comme $2 \times 7 = 14$ on additionne $1 + 4$ et donc on renvoie 9),
   - `mult2(9)` renvoie 9 (comme $2 \times 9 = 18$ on additionne $1 + 8$ et donc on renvoie 9),

```c
// sinon la somme des chiffres de 2*c
int mult2(int c)
{
    assert(c >= 0 && c <= 9);
    int d = 2 * c;
    if (d >= 10)
    {
        return (d % 10 + 1);
    }
    else
    {
        return d;
    }
}

```

2. Pour vérifier que la fonction `mult2` est totalement correcte, dix tests suffisent, lesquels et pourquoi ? Donner ces dix tests sous forme d'instructions `assert`.

   La fonction `mult2` ne prend que dix entrées possibles (les dix chiffres $0, 1, \ldots, 9$), donc pour vérifier qu'elle est correcte il suffit de tester la valeur renvoyée pour ces dix tests à l'aide des instructions `assert` suivantes :

```c
    assert(mult2(0) == 0);
    assert(mult2(1) == 2);
    assert(mult2(2) == 4);
    assert(mult2(3) == 6);
    assert(mult2(4) == 8);
    assert(mult2(5) == 1);
    assert(mult2(6) == 3);
    assert(mult2(7) == 5);
    assert(mult2(8) == 7);
    assert(mult2(9) == 9);
```

3. L'algorithme de Luhn consiste à faire la somme des chiffres du numéro de carte de crédit en utilisant au préalable la fonction `mult2` ci-dessus sur les chiffres de rang pair (c'est à dire en partant de la fin du nombre, le 2e chiffre, le 4e chiffre, ...). Si la somme obtenue est divisible par 10 alors le numéro est valide. Par exemple :
   - pour `267` on doit faire `2 + mult2(6) + 7` ce qui donne `2+3+7 = 12` et donc ce numéro est invalide.
   - pour `15782`, on doit faire la somme `1 + mult2(5) + 7 + mult2(8) + 2`, ce qui donne : `1+1+7+7+2 = 18` et donc ce numéro est invalide.
   - pour `124586`, on doit faire la somme `mult2(1) + 2 + mult2(4) + 5 + mult2(8) + 6`, ce qui donne : `2+2+8+5+7+6 = 30` et donc ce numéro est valide puisque `30` est divisible par `10`.

   a) Vérifier que le numéro `4762` est valide.

   On calcule : `mult2(4) + 7 + mult2(6) + 2 = 8 + 7 + 3 + 2`, on obtient 20 qui est bien divisible par 10 et donc `4762` est un numéro valide.

   b) Ecrire une fonction de prototype `bool valide(int n)` qui prend en entrée un numéro de carte de crédit et renvoie un booléen indiquant si ce numéro est valide.

```c
bool valide(int num)
{
    int spair = 0;
    int simpair = 0;
    bool impair = true;
    int chiffre;
    while (num != 0)
    {
        chiffre = num % 10;
        num = num / 10;
        if (impair)
        {
            simpair += chiffre;
        }
        else
        {
            spair += mult2(chiffre);
        }
        impair = !impair;
    }
    return (simpair + spair) % 10 == 0;
}

```

   c) Ecrire une fonction qui prend en entrée un entier `n` et détermine quel chiffre ajouter à droite de ce nombre de façon à ce que le nombre ainsi formé soit un numéro de carte de crédit valide. Par exemple pour 543, cette fonction renvoie le chiffre 9 car 5439 est un numéro de carte de crédit valide (mult2(5) + 4 + mult2(3) + 9 = 1 + 4 + 6 + 9 = 20) .

```c
int cree_valide(int n)
{
    for (int c = 0; c < 10; c++)
    {
        if (valide(10 * n + c))
        {
            return c;
        }
    }
}
```