Devoir surveillé d'informatique

Consignes

- Les programmes demandés doivent être écrits en C et on suppose que les librairies standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.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 : *chercher les erreurs*

1. Les fonctions ci-dessous contiennent une ou plusieurs erreurs et/ou ne respectent pas leur spécification (donnée en commentaire). Dans chaque cas, expliquer les erreurs commises et les corriger de façon à ce que la fonction obéisse à sa spécification.

   a) Fonction `harmonique`

```c
// Calcule la somme des inverses des entiers jusqu'à n
// Par exemple harmonique(3) renvoie 1 + 1/2 + 1/3
double harmonique(int n){
    double sh = 0;
    for (int i = 0; i < n; i = i + 1)
    {
        sh = sh + 1 / i;
    }
    return sh;}
```

- A la ligne 6, on ne doit pas commencer à l'indice 0 (sinon on effectue une division par zéro) ni s'arrêter à l'indice $(n-1)$ (on ne respecte pas la specification). La ligne correcte est donc `for (int i=1; i<n+1; i = i + 1)`
- A la ligne 8, `1/i` est une division entière car les deux opérandes sont des entiers. Le résultat est donc 0 (même si ensuite on a une conversion implicite de type vers `double` de façon à additionner avec `sh`). On doit donc pour forcer la division décimale en changeant cette ligne en `sh = sh + 1.0/i;` ou encore avec une conversion explicite : `sh = sh + 1/(double)i;`

   b) Fonction `tous_egaux`

```c
// Renvoie true si tous les éléments de tab sont identiques
// Par exemple si test={2, 2, 2, 2} alors tab(test,4) renvoie true
bool tous_egaux(int tab[], int size){
    for (int i = 0; i < size; i = i + 1)
    {
        if (tab[i] != tab[i + 1])
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}
```

• la ligne 7 provoque un accès en dehors des limites du tableau car i a pour valeur maximale `size-1`. On doit donc modifier la lignes 5 en `for (int i=0; i<size-1; i = i + 1)`
• Le bloc du `else` ligne 11 à 14 renvoie `true` dès que les deux premiers éléments sont égaux. Il ne faut le faire qu'après avoir parcouru tout le tableau. Le `if` ne devrait donc pas avoir de bloc `else`, et le `return true` doit être positioné après le bloc de la boucle `for`

c) Fonction `cree_tab_entiers`

```
1  // Renvoie un pointeur vers un tableau contenant les entiers de 0 à n-1
2  int *cree_tab_entiers(int n){
3      int tab_entiers[n];
4      for (int i = 0; i < n; i++)
5      {
6          tab_entiers[i] = i;
7      }
8      return &tab_entiers;}
```

• le tableau `tab_entiers` de la ligne 4 est alloué sur la pile car c'est une variable locale à la fonction. On doit l'allouer sur le tas de façon à conserver cette zone mémoire en quittant la fonction. On doit donc remplacer cette ligne par `int *tab_entiers = malloc(sizeof(int)*size);`
• `tab_entiers` est un tableau, donc un pointeur vers l'adresse de son premier élément. On doit donc simplement écrire `return tab_entiers;`

d) Fonction `syracuse`

```
1  // Ne renvoie rien et remplace n par n/2 si n est pair et 3n+1 si n est impair
2  void syracuse(int n){
3      if (n % 2 = 0)
4      {
5          n = n / 2;
6      }
7      else
8      {
9          n = 3 * n + 1;
10     }}
```

• Les variables en C sont passés par valeur, donc cette fonction ne modifie pas `n`. On doit si on veut respecter la spécification passer un pointeur vers `n`. La ligne 2 est donc `void syracuse(int *n)` et on doit remplacer `n` par `*n` dans le reste de la fonction. L'appel à cette fonction si `n` est déclaré comme un entier se fera avec `syracuse(&n)` de façon à passer l'adresse de `n`.
• En C, le test d'égalité est `==`, la ligne 4 est donc `if (n%2 == 0)`.

2. Les programmes suivants, produisent une erreur à l'exécution. Expliquer l'origine du problème et apporter les corrections nécessaires

a) Affichage d'une adresse

```
1  // Programme qui affiche l'adresse de la variable p
2  int main(){
3      int *p;
4      *p = 42;
5      printf("%p", p);}
```

La variable `p` est un pointeur *non initialisé* vers un entier, aucune zone mémoire n'a été réservé pour sa valeur. On doit donc lorsqu'on déclare cette variable lui alloué un emplacement mémoire avec `malloc`. La ligne 4 est donc `int *p = malloc(sizeof(int));`

b) Calcul de la somme de deux entiers

```c
// Programme qui demande deux entiers puis affiche leur somme
int main(){
    int a, b;
    printf("a=");
    scanf("%d", a);
    printf("b=");
    scanf("%d", b);
    printf("Somme = %d", a + b);}
```

La fonction `scanf` va modifier la valeur de la variable donnée en argument (ici `a` puis `b`), donc elle prend en argument des pointeurs vers ces valeurs (sinon elle ne pourrait pas les modifier), la ligne 6 est donc `scanf("%d", &a);`

❑ Exercice 2 : *Pointeurs*

1. Compléter le tableau suivant, qui donne l'état des variables au fur et à mesure des instructions données dans la première colonne (on a indiqué par ✖ une variable non encore déclaré.)

instructions	a	b	p	q
`int a = 14;`	14	✖	✖	✖
`int b = 42;`	14	42	✖	✖
`int *p = &a;`	14	42	&a	✖
`int *q = &b;`	14	42	&a	&b
`*p = *p + *q ;`	56	42	&a	&b
`*q = *p - *q ;`	56	14	&a	&b
`*p = *p - *q ;`	42	14	&a	&b

2. Ecrire une fonction en C qui prend en argument deux pointeurs vers des entiers, ne renvoie rien et échange les valeurs de ces deux entiers *sans utiliser de variable temporaire*.

```c
void echange(int *p, int *q)
{
    *p = *p + *q;
    *q = *p - *q;
    *p = *p - *q;
}
```

3. Compléter le programme suivant en écrivant l'appel à la fonction `echange` afin d'échanger les valeurs des entiers `n` et `m`

```c
    int n = 55, m = 12;
    echange(&n, &m);
```

❑ Exercice 3 : *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
int valeur_absolue(int n)
{
    if (n < 0)
        return -n;
    return n;
}
```

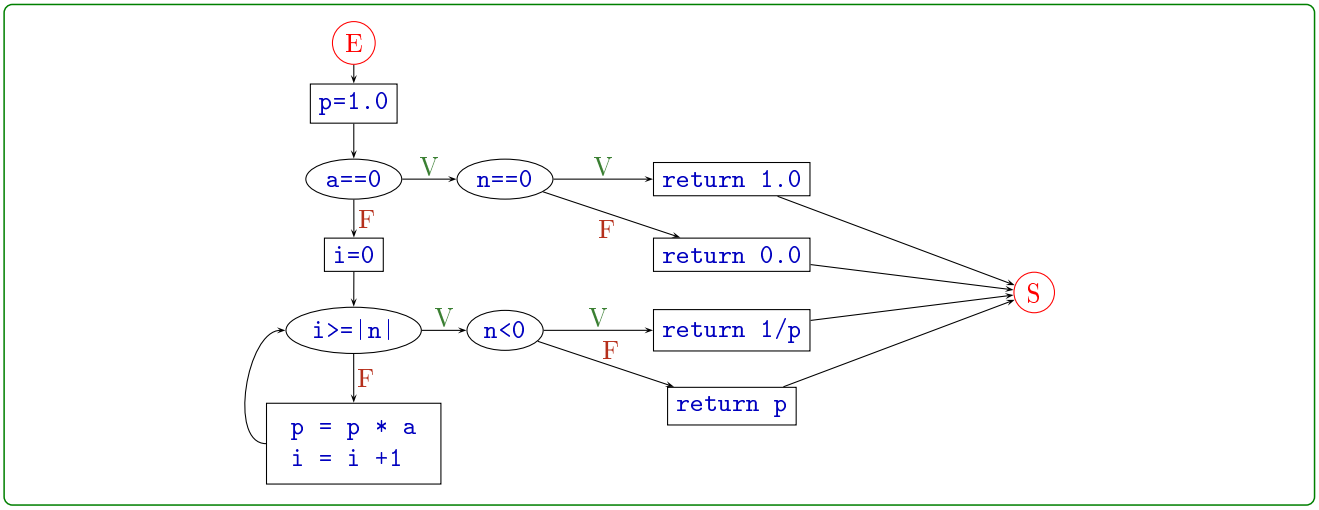
2. Ecrire une fonction `puissance` qui prend en argument un flottant (type `double`) a et un entier n et renvoie a^n. On rappelle que pour $a \in \mathbb{R}^*$, $n \in \mathbb{Z}$:

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

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

```
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 4** : *Annagrammes*

Deux mots *de même longueur* sont anagrammes l'un de l'autre lorsque l'un est formé en réarrangeant les lettres de l'autre. Par exemple :

— *niche* et *chien* sont des anagrammes.

— *epele* et *pelle*, ne sont pas des anagrammes, en effet bien qu'ils soient formés avec les mêmes lettres, la lettre *l* ne figure qu'à un seul exemplaire dans *epele* et il en faut deux pour écrire *pelle*.

Le but de l'exercice est d'écrire une fonction en C qui renvoie `true` si les deux chaines données en argument sont des anagrammes et `false` sinon. On suppose par facilité que les chaines sont constitués uniquement de lettres majuscules.

1. Ecrire une fonction `tableau_egaux` qui prend en argument deux tableaux ainsi que leur taille et renvoie `true` lorsque ces deux tableaux ont un contenu et une longueur identique et `false` sinon.

```
bool tab_egaux(int tab1[],int size1, int tab2[], int size2)
{
    if (size1!=size2)
    {return false;}
    for (int i=0;i<size1;i++)
    {
        if (tab1[i]!=tab2[i])
        {return false;}
    }
    return true;
}
```

2. Ecrire une fonction `nb_lettres` qui prend en argument une chaine de caractères `chaine` et renvoie un tableau d'entiers `tab` de longueur 26 de sorte que `tab[i-1]` contienne le nombre de fois où la ième lettre de l'alphabet apparait dans `chaine`. Par exemple tab[0], doit contenir le nombre de fois où la lettre *a* apparait dans le mot.

```
int * nb_lettres(char *chaine)
{
    int i = 0;
    int index;
    int * tab = malloc(sizeof(int)*26);
    for (int i=0; i<26; i++)
    {
        tab[i] = 0;
    }
    while (chaine[i]!='\0')
    {
        index = (int) chaine[i] - 65;
        tab[index] = tab[index] + 1;
        i = i + 1;
    }
    return tab;
}
```

3. En supposant les deux fonctions précédentes correctement écrites, on propose le code suivant pour la fonction `anagrammes` qui teste si les deux chaines données en argument sont des anagrammes :

```
bool anagrammes(char * chaine1, char *chaine2)
{
   return tab_egaux(nb_lettres(chaine1),26,nb_lettres(chaine2),26);
}
```

Cette fonction renvoie le résultat attendu mais pose un problème, lequel ? Expliquer et proposer une correction.

La fonction `nb_lettres` renvoie un pointeur vers une zone mémoire allouée sur le tas. Dans la fonction `anagrammes`, on ne conserve pas de référence sur cette zone (elle est directement utilisée sur la fonction `tab_egaux`). Donc, cette zone mémoire n'est pas libérable par un `free` puisqu'on n'a pas de référence vers elle. Chaque appel à la fonction `anagrammes` entraine donc une fuite mémoire. On peut proposer la fonction suivante comme correction, où on garde des références vers les tableaux crées par `nb_lettres` de façon à libérer l'espace mémoire alloué.

```
bool anagrammes(char * chaine1, char *chaine2)
{
   int *tab1 = nb_lettres(chaine1);
   int *tab2 = nb_lettres(chaine2);
   bool anag;
   if (tab_egaux(tab1,26,tab2,26))
   {
       anag = true;
   }
   else
   {
       anag = false;
   }
   free(tab1);
   free(tab2);
   return anag;

}
```

❑ Exercice 5 : *tri à bulles*

Le tri à bulles est un algorithme de tri qui parcourt le tableau à l'aide d'un indice i de la fin vers le début. Pour chacun de ces indices i, on parcourt la partie du tableau allant de l'indice 0 à l'indice $i-1$ et si les deux éléments consécutifs situés aux indices j et $j+1$ ne sont pas dans l'ordre croissant, on les échange. Par exemple sur le tableau `{7, 2, 9, 5, 3}`

- $i = 4$ (on rappelle que i parcourt de la fin vers le début)
  - $j = 0$, donc on échange `tab[0]` et `tab[1]` car ils ne sont pas dans l'ordre croisant : `{2, 7, 9, 5, 3}`
  - $j = 1$, pas d'échange (7 et 9 sont en ordre croissant)
  - $j = 2$, échange car 9 et 5 ne sont pas dans l'ordre croissant : `{2, 7, 5, 9, 3}`
  - $j = 3$, échange et on obtient `{2, 7, 5, 3, 9}`
- $i = 3$, cette fois on parcourt avec $j = 0$ jusqu'à 2, en effet à l'étape précédente le plus grand élément du tableau se retrouve forcement en dernière position. On obtient en fin de parcours : `{2, 5, 3, 7, 9}`
- $i = 2$, à la fin de cette itération on obtient `{2, 3, 5, 7, 9}`

1. Faire fonctionner cet algorithme à la main sur le tableau `{11, 2, 5, 13, 8, 4}` et donner l'état du tableau à la fin de chaque itération de l'indice j pour j variant de 0 à i en recopiant et complétant le

tableau suivant :

i	valeurs contenu dans le tableau à la fin de l'itération d'indice i.
5	{2,5,11,8,4,13}
4	{2,5,8,4,11,13}
3	{2,5,4,8,11,13}
2	{2,4,5,8,11,13}
1	{2,4,5,8,11,13}

2. Donner un exemple de tableau de longueur 5, *non trié initialement* qui sera entièrement trié après le premier tour de boucle de l'indice j (c'est à dire pour $i = 4$).

   Si les éléments sont déjà triés à l'exception de l'élément maximal, la seule itération de j pour $j = 0$ jusqu'à $i = 4$, suffit à ramener cet élément en fin tableau. Donc par exemple le tableau `{ 2, 99, 3, 4, 5 }` sera entièrement trié dès que $i = 3$..

3. Donner un exemple de tableau qui ne sera trié qu'à la fin de toutes les itérations de l'indice i.

   Si le tableau a son élément minimal tout à la fin, alors cet élément se déplace vers le début du tableau une fois par itération de l'indice i, donc le tableau ne sera trié qu'à la toute fin de l'algorithme. On peut donner l'exemple du tableau `{ 50, 49, 48, 47, 1 }`

4. On veut maintenant écrire cet algorithme en C, en effectuant le tri dans une copie du tableau. On suppose déjà écrite la fonction `echange` qui prend en argument un tableau et deux indices ne renvoie rien et échange les éléments situés à ces deux indices.

   a) Ecrire un fonction `copie_tab` qui prend en argument un tableau ainsi que sa taille et renvoie un pointeur vers une copie de ce tableau

```c
int* copie_tab(int tab[],int size)
{
    int *ctab = malloc(sizeof(int)*size);
    for (int i=0; i<size; i++)
    {
        ctab[i] = tab[i];
    }
    return ctab;
}
```

   b) Ecrire une fonction `tri_bulles` qui prend en argument un tableau `tab` ainsi que sa taille, ne modifie pas ce tableau et renvoie un pointeur vers un tableau contenant les éléments du tableau `tab` triés dans l'ordre croissant.

```c
int* tri_bulles(int tab[],int size)
{
    int *copie = copie_tab(tab,size);
    for (int i=size-1;i>0;i=i-1)
    {
        for (int j=0; j<i; j++)
        {
            if (copie[j]>copie[j+1])
            {
                echange(copie,j,j+1);
            }
        }
        affiche(copie,size);
        printf("\n");
    }
    return copie;
}
```

   c) Afin de tester cette fonction on a écrit le programme principal suivant :

```c
int main()
{
    int tab[6] = {5, 11, 8, 9, 4, 6};
    int *tab_trie = trie_bulles(tab,6);
    for (int i=0; i<6; i++)
    {printf("%d ", tab_trie[i]);}
    printf("\n");
}
```

Quelle instruction est manquante dans ce programme ? Quelle option de compilation signalerait le problème lors de l'exécution ?

On doit libérer la mémoire allouée, l'instruction manquante est donc `free(tab_trie);` qu'on peut placer dès qu'on a plus besoin du tableau trié (donc ici après la boucle `for` d'affichage). Le problème serait signalé lors de l'exécution si on on compile avec l'option `fsanitize = address`.