Devoir surveillé d'informatique

Consignes

- Les programmes demandés doivent être écrits en C ou en OCaml. Dans le cas du 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 donne ci-dessous l'algorithme d'exponentiation rapide en version itérative :

Algorithme : Exponentiation rapide

Entrées : $a \in \mathbb{R}, n \in \mathbb{N}$

Sorties : a^n

$p \leftarrow 1$
tant que $n \neq 0$ **faire**
    si n *est impair* **alors**
        $p \leftarrow p \times a$
    fin
    $a \leftarrow a * a$
    $n \leftarrow \lfloor \frac{n}{2} \rfloor$
fin
return p

1. Donner les valeurs successives prises par les variables a, n et p si on fait fonctionner cet algorithme avec $a = 2$ et $n = 13$. On pourra recopier et compléter le tableau suivant et donner les valeurs de a et de p sous la forme de puissance de 2 :

	a	n	p
valeurs initiales	2	13	1
après un tour de boucle	2^2	6	2
après deux tours de boucle	2^4	3	2
après trois tours de boucle	2^8	1	2^5
après quatre tours de boucle	2^{16}	0	2^{13}

2. Donner une implémentation de cet algorithme en langage C sous la forme d'une fonction `exp_rapide` de signature `double exp_rapide(double a, int n)`. 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
double exp_rapide(double a, int n)
{ // pour n positif, renvoie a puissance n
    assert(n >= 0);
    double p = 1.0;
    while (n != 0)
    {

        if (n % 2 == 1)
        {
            p = p * a;
        }
        a = a * a;
        n = n / 2;
    }
    return p;
}
```

3. Prouver que cet algorithme termine.

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

1. $n \in \mathbb{N}$ par précondition.
2. n reste positif par condition d'entrée dans la boucle.
3. n décroit strictement car n est divisé par 2 lors de chaque passage dans la boucle.

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

4. Prouver que cet algorithme est correct. En notant a_0 (resp. n_0) la valeur initiale de a (resp. n), on pourra prouver l'invariant $p \times a^n = a_0^{n_0}$.

On note, a_0 la valeur initiale de a et n_0 la valeur initiale de n, montrons que la propriété I : « $p \times a^n = a_0^{n_0}$ » est un invariant de boucle.

1. Avant d'entrée dans la boucle $p = 1$, $a = a_0$ et $n = n_0$ donc $p \times a^n = a_0^{n_0}$ et I est vérifiée.
2. On suppose I vérifié à l'entrée de la boucle et on note a' (resp. n', resp. p') les valeurs prises par a (resp. n, resp. p) au tour de boucle suivant, alors :

   — Si n est paire, $n' = n/2$, $p' = p$ et $a' = a^2$ donc
   $p' \times a'^{n'} = p \times (a^2)^{n/2}$
   $p' \times a'^{n'} = p \times a^n$
   et puisque I était vraie en entrée de boucle, $p' \times a'^{n'} = a_0^{n_0}$

   — Sinon, $n' = (n-1)/2$, $p' = p \times a$ et $a' = a^2$ donc
   $p' \times a'^{n'} = p \times a \times (a^2)^{(n-1)/2}$
   $p' \times a'^{n'} = p \times a^n$
   et puisque I était vraie en entrée de boucle, $p' \times a'^{n'} = a_0^{n_0}$

En sortie de boucle, puisque $n = 0$, cet invariant donne $p \times a^0 = a_0^{n_0}$ et donc $p = a_0^{n_0}$ et donc l'algorithme est correcte.

5. Donner une implémentation récursive de l'algorithme d'exponentiation rapide en OCaml sous la forme d'une fonction `exp_rapide float -> int -> float`

```
let rec exp_rapide a n =
  if n=0 then 1.0 else
    let temp = exp_rapide a (n/2) in
    if (n mod 2=0) then temp*.temp else a*.temp*.temp;;
```

❑ 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 déclarée.)

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 `echange` 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;
    int m = 12;
```

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

❑ Exercice 3 : *Quelques expression en OCaml*

Pour chacune des expressions ci-dessous, indiquer son type et sa valeur lorsqu'elle s'évalue sans erreur. Sinon indiquer la cause de l'erreur rencontrée.

1. `let n = 24 mod 7;;`

   `n` est un entier (type `int`) valant 3 (reste dans la division euclidienne de 24 par 7.)

2. `let perimetre = 4 *. 2.5;;`

   L'évaluation donne une erreur car l'opérateur `*.` est la multiplication entre deux opérandes de type `float`, ici l'une des opérandes (4) est un `int`.

3. `let v = 2.0**10;;`

   En OCaml l'opérateur `**` est l'exponentiation, mais il n'est défini que pour deux opérandes de type `float`, on obtient de nouveau une erreur puisque l'une des opérandes est entière. Pour calculer 2^{10}, il faudrait écrire `let v = 2.0**10.;;`

4. `let at = '@' in print_char at;;`

   Cette expression s'évalue correctement (`at` est bien de type `char` car entre simple quotes), elle renvoie `()` de type `unit` car c'est un affichage.

5. `let coucou = let message = "Bonjour " + "tout le monde" in print_string message;;`

   L'opérateur `+` est l'addition de deux entiers, on obtient donc une erreur, l'opérateur de concaténation entre deux chaines de caractères est `^`.

6. `let peri = let cote = 5 in 4*cote;;`

   L'expression s'évalue correctement et vaut l'entier `20`.

7. `let k = if 2=1+1 then 'A' else 'B';;`

   L'expression s'évalue correctement et vaut `'A'` (type `char`).

8. `let rec fact n = if n=0 then 1 else n* fact (n-1);;`

   L'expression s'évalue correctement, c'est une fonction (type `fun`) `fact : int -> int` (qui calcule la factorielle de `n`)

❑ **Exercice 4** : *Un tableau qui connait sa taille*
En C, on propose le type structuré suivant afin de représenter un « tableau d'entiers qui connaît sa taille » :

```c
struct tableau_s
{
    int taille;
    int *valeurs;
};
typedef struct tableau_s tableau;
```

La champ `taille` contient la taille du tableau et le champ `valeurs` est un pointeur vers une zone mémoire contenant la liste des valeurs du tableau.

1. Ecrire une fonction `somme` de signature `int somme(tableau t)` qui renvoie la somme des valeurs contenus dans `t`

```c
int somme(tableau t)
{
    int s = 0;
    for (int i = 0; i < t.taille; i++)
    {
        s += t.valeurs[i];
    }
    return s;
}
```

2. On veut écrire une fonction `cree_tableau` de signature `tableau cree_tableau(int val, int taille)` qui renvoie un `tableau` de taille `taille` dont toutes les valeurs sont initialisées à `val`. La solution proposée ci-dessous (appelée `cree_tableau_bug`) compile sans erreur et sans avertissement (avec les options `-Wall` et `-Wextra`) mais ne fonctionne pas correctement (on obtient une erreur à l'exécution ou les valeurs présentes dans le tableau ne sont pas égales à `val`).

```
1  tableau cree_tableau_bug(int val, int taille)
2  {
3      tableau t;
4      t.taille = taille;
5      int tab[taille];
6      for (int i = 0; i < taille; i++)
7      {
8          tab[i] = val;
9      }
10     t.valeurs = tab;
11     return t;
12 }
```

Expliquer ce comportement en utilisant vos connaissances sur le modèle mémoire du langage C (on pourra illustrer par un schéma)

Le tableau `tab` déclaré à la ligne 5 est stocké sur la pile car c'est une variable locale à la fonction, aussi `t.valeurs` pointe sur la pile dans une zone mémoire qui sera libéré à la sortie de la fonction car le contexte d'appel de la fonction (et donc les variables locales) est alors désalloué. Pour éviter ce problème, on doit allouer sur le tas à la ligne 5 à l'aide d'une instruction `malloc`. Cela permettra de `t.valeurs` vers un zone mémoire qui sera conservée intacte à la sortie de la fonction.

3. Proposer une version correcte de la fonction `cree_tableau`.

```
tableau cree_tableau(int valeur_initiale, int taille)
{
    tableau s;
    s.taille = taille;
    s.valeurs = malloc(sizeof(int) * taille);
    for (int i = 0; i < taille; i++)
    {
        s.valeurs[i] = valeur_initiale;
    }
    return s;
}
```

4. Le *crible d'Erastothène* est un algorithme permettant de trouver tous les nombres premiers inférieurs à un entier N donné. Il procède en parcourant la liste des entiers de 2 à la racine carrée de N en supprimant les multiples des nombres non encore éliminés rencontrés. Par exemple pour trouver les nombres premiers inférieurs à 20, on part de la liste des entiers de 2 à 20 :

   — 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 et on barre les multiples de 2 (excepté 2 lui-même).

   — 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, 19, ~~20~~ on barre les multiples de 3 (excepté 3)

   — 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~ comme $5 > \sqrt{20}$, l'algorithme s'arrête.

   Les nombres premiers inférieurs à 20 sont les nombres non barrés c'est à dire $\{2, 3, 5, 7, 11, 13, 17, 19\}$.
   Afin d'implémenter cet algorithme en langage C, on propose d'utiliser un tableau de booléens `premiers` de taille $N+1$ et de mettre `premiers[i]` à `false` lorsqu'on barre `i`. Donc on initialise le tableau à `true`

sauf `premiers[0]` et `premiers[1]` qui valent `false` puis on parcourt ce tableau à l'aide d'un indice `i` (de 2 à $\sqrt{N}$), si `premiers[i]` est à `true` alors on met tous les `premiers[k]` où `k` est un multiple de `i` strictement plus grand que `i` à `false`. Ce parcours s'arrête dès que l'entier `i` est supérieur à $\sqrt{N}$. Ecrire cette implémentation sous la forme d'une fonction `crible` de signature `bool *crible(int n)` qui prend en argument un entier `n` et renvoie un tableau de booléens `premiers` tel que `premiers[i]` vaut `true` si et seulement si `i` est premier. On supposera *déjà écrite* une fonction `sqrt` de signature `int sqrt(int n)` qui renvoie la partie entière de $\sqrt{\texttt{n}}$.

```c
bool *crible(int n)
{
    bool *premiers = malloc(sizeof(bool) * (n + 1));
    premiers[0] = false;
    premiers[1] = false;
    int k;
    for (int i = 2; i <= n; i++)
    {
        premiers[i] = true;
    }
    for (int i = 2; i < isqrt(n) + 1; i++)
    {
        if (premiers[i])
        {
            k = 2;
            while (k * i <= n)
            {
                premiers[k * i] = false;
                k = k + 1;
            }
        }
    }
    return premiers;
}
```

5. En utilisant la question précédente, écrire une fonction `nombres_premiers` de signature `tableau nombres_premiers(int n)` qui prend en argument un entier `n` et renvoie un `tableau` contenant les nombres premiers inférieurs ou égaux à `int n`. Par exemple, si `n=20`, cette fonction renvoie un tableau `t`, telle que `t.taille=8` et contenant les valeurs $\{2, 3, 5, 7, 11, 13, 17, 19\}$.

```c
1  tableau nombres_premiers(int n)
2  {
3      bool *premiers = crible(n);
4      int nb = 0;
5      tableau t;
6      for (int i = 0; i <= n; i++)
7      {
8          if (premiers[i])
9          {
10             nb += 1;
11         }
12     }
13     t.taille = nb;
14     t.valeurs = malloc(sizeof(int) * nb);
15     for (int i = 0; i <= n; i++)
16     {
17         if (premiers[i])
18         {
19             t.valeurs[nb] = i;
20         }
21     }
22     free(premiers);
23     return t;
24 }
```

6. Expliquer rapidement pourquoi la fonction `nombres_premiers` (qui utilise `crible`) doit nécessairement contenir une instruction `free`.

   La fonction `nombres_premiers` fait appel à `crible` qui alloue sur le tas un tableau de booléens, en dehors de la fonction `nombres_premiers` on ne dispose pas de référence vers ce tableau, il doit donc être libéré depuis cette fonction (ligne 22 du programme précédent).

❑ **Exercice 5** : *Implémentation des entiers par représentation binaire*

On rappelle qu'en C, le type `uint64_t` (disponible dans `stdint.h` qu'on suppose déjà importée dans la suite de l'exercice) représente des entiers *positifs* (non signés) sur 64 bits. D'autre part on rappelle que le spécificateur de format permettant d'afficher un entier de type `uint64_t` est `%lu`.

1. A propos du format `uint64_t`.

   a) Donner l'intervalle d'entiers représentable avec ce format.

   Les entiers représentables avec ce format sont $[\![0; 2^{64} - 1]\!]$.

   b) En compilant puis en exécutant le programme suivant sur un ordinateur (les librairies <stdio.h> et <stdint.h> sont supposées importées) :

```c
int main()
{
    uint64_t a = 0;
    a = a - 1;
    printf("a= %lu\n", a);
}
```

   on a obtenu l'affichage suivant dans le terminal : `a= 18446744073709551615`. Expliquer cet affichage, s'agit-il d'un comportement indéfini ?

Comme `a` est un entier non signé initialisé à 0, l'instruction `a = a -1` est un dépassement de capacité. Ce n'est pas un comportement indéfini, sur les entiers non signés les calculs sont faits modulo le plus grand entier représentable plus un et donc ici on obtient donc $2^{64} - 1$.

2. Représentation des ensembles.
On utilise à présent les entiers au format `uint64_t` afin de représenter des ensembles. A chaque entier écrit en base 2 on associe l'ensemble dont les éléments sont les positions des bits égaux à 1. Par exemple :
   — L'entier $\overline{11001}^2(=\overline{25}^{10})$ a des bits égaux à 1 aux positions 0,3 et 4 et donc représente l'ensemble $\{0, 3, 4\}$.
   — L'entier $\overline{10000000}^2(=\overline{128}^{10})$ a un seul bit égal à 1 en position 7 et donc représente l'ensemble $\{7\}$.
   — L'ensemble $\{1, 5\}$ est représenté par l'entier ayant des bits égaux à 1 en position 1 et 5, c'est à dire $\overline{100010}^2 = \overline{34}^{10}$.

a) Quels sont les ensembles représentables avec ce codage avec des entiers au format `uint64_t` ?

Les ensembles répresentables sont les parties de $[\![0; 63]\!]$

b) Donner l'écriture en base 10 de l'entier représentant l'ensemble $\{2, 7\}$

L'entier représentant $\{2, 7\}$ est $\overline{10000100}^2 = \overline{132}^{10}$.

c) Quel est l'ensemble codé par l'entier $\overline{76}^{10}$?

$\overline{76}^{10} = \overline{1001100}^2$ et donc code l'ensemble $\{2, 3, 6\}$.

d) Donner la caractérisation des ensembles représentés par une puissance exacte de 2 (on ne demande pas de justification).

Les ensembles représentés par une puissance exacte de 2 sont les singletons.

e) Ecrire une fonction `appartient` de signature `bool appartient(uint64_t s, int e)` qui prend en argument un entier `s` (type `uint64_t`) représentant un ensemble et un entier `e` et renvoie `true` si `e` appartient à l'ensemble représenté par `s` et `false sinon`. Par exemple puisque l'ensemble $\{1, 5\}$ est codé par 34, `appartient(34,1)` doit renvoyer `true` tandis que `appartient(34,2)` doit renvoyer `false`.

On utilise l'algorithme des divisions successives de façon à trouver le bit de rang `e`, on renvoie `true` si ce bit est à 1 et `false sinon`

```c
bool appartient(uint64_t s, int e)
{
    while (e != 0)
    {
        s = s / 2;
        e = e - 1;
    }
    return (s % 2 == 1);
}
```

Les opérations bit à bit ne sont pas au programme, mais ils permettent ici d'écrire une solution bien plus concise. L'opérateur `>>e` décale les bits de s de e rang vers la droite et `& 1` permet de récupérer le dernier bit.

```c
bool appartient_bb(uint64_t s, int e)
{
    return s >> e & 1 == 1;
}
```

f) Ecrire une fonction `encode` en C de signature `uint64_t encode(bool tab[])`, qui prend en argument un tableau `tab` de 64 booléens et renvoie l'entier au format `uint64_t` qui représente l'ensemble dont les éléments sont les entiers `i` tels que `tab[i]=true`. Par exemple, si `tab` est le tableau de booléens de taille 64 ne contenant que des `false` sauf `tab[3]` et `tab[10]` qui valent `true` alors, `encode(tab)` doit renvoyer l'entier qui représente l'ensemble `{3, 10}`.

```c
uint64_t encode(bool tab[])
{
    uint64_t res = 0;
    uint64_t poids = 1;
    for (int i = 0; i < 64; i++)
    {
        if (tab[i])
        {
            res += poids;
        }
        poids = poids * 2;
    }
    return res;
}
```

g) Ecrire une fonction `decode` de signature `bool *decode(uint64_t n)`, qui prend en argument un entier `n` au format `uint64_t` et renvoie l'ensemble qu'il représente sous la forme d'un tableau `tab` de 64 booléens tels que `tab[i]=true` si et seulement si `i` appartient à l'ensemble représenté par `n`. Par exemple `decode(34)` doit renvoyer un tableau `tab` de booléens dont toutes les valeurs sont `false` sauf `tab[1]` et `tab[5]` qui valent `true`.

```c
bool *decode(uint64_t n)
{
    bool *tab = malloc(sizeof(bool) * 64);
    for (int i = 0; i < 64; i++)
    {
        tab[i] = (n%2 ==1);
        n =n /2;
    }
    return tab;
}
```