# 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 :** $a^n$

```
1 p ← 1
2 tant que n ≠ 0 faire
3 |   si n est impair alors
4 |   |   p ← p × a
5 |   fin
6 |   a ← a * a
7 |   n ← ⌊n/2⌋
8 fin
9 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 | ... | ... | ... |
| après deux tours de boucle | ... | ... | ... |
| après trois tours de boucle | ... | ... | ... |
| après quatre tours de boucle | ... | ... | ... |

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`.
3. Prouver que cet algorithme termine.
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 $p \times a^n = a_0^{n_0}$.
5. Donner une implémentation récursive de l'algorithme d'exponentiation rapide en OCaml sous la forme d'une fonction `exp_rapide float -> int -> float`.

## ❑ 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 | ... | ... | ... |
| `int *p = &a;` | 14 | ... | &a | ... |
| `int *q = &b;` | 14 | ... | &a | ... |
| `*p = *p + *q ;` | ... | ... | ... | ... |
| `*q = *p - *q ;` | ... | ... | ... | ... |
| `*p = *p - *q ;` | ... | ... | ... | ... |

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

```
1     int n = 55;
2     int m = 12;
```

❑ **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;;`
2. `let perimetre = 4 *. 2.5;;`
3. `let v = 2.0**10;;`
4. `let at = '@' in print_char at;;`
5. `let coucou = let message = "Bonjour " + "tout le monde" in print_string message;;`
6. `let peri = let cote = 5 in 4*cote;;`
7. `let k = if 2=1+1 then 'A' else 'B';;`
8. `let rec fact n = if n=0 then 1 else n* fact (n-1);;`

❑ **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 » :

```
1  struct tableau_s
2  {
3      int taille;
4      int *valeurs;
5  };
6  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`.
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`).

```c
tableau cree_tableau_bug(int val, int taille)
{
    tableau t;
    t.taille = taille;
    int tab[taille];
    for (int i = 0; i < taille; i++)
    {
        tab[i] = val;
    }
    t.valeurs = tab;
    return t;
}
```

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

3. Proposer une version correcte de la fonction `cree_tableau`.
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 :
   — $\underline{2}$, 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, $\underline{3}$, ~~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~~, $\underline{5}$, ~~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 $\sqrt{\texttt{n}}$.
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 $\{2, 3, 5, 7, 11, 13, 17, 19\}$.
6. Expliquer rapidement pourquoi la fonction `nombres_premiers` (qui utilise `crible`) doit nécessairement contenir une instruction `free`.

❑ **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.
   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 ?

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 ainsi représentables ?
   b) Donner l'écriture en base 10 de l'entier représentant l'ensemble $\{2, 7\}$
   c) Quel est l'ensemble codé par l'entier $\overline{76}^{10}$ ?
   d) Donner la caractérisation des ensembles représentés par une puissance exacte de 2 (on ne demande pas de justification).
   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`.
   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}`.
   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`.