❑ **Exercice 1** : *Exponentiation rapide*

1. Rappeler l'algorithme d'exponentiation rapide et donner les étapes du calcul de a^{20} avec cet algorithme.
2. Ecrire une implémentation *récursive* de cet algorithme en OCaml permettant de calculer a^n avec a et n entiers.
3. On donne l'implémentation *itérative* suivante de cet algorithme en langage C :

```c
uint32_t exprap(uint32_t a, unsigned n)
{
    uint32_t r = 1;
    uint32_t  k = a;
    unsigned  p = n;
    while (p != 0)
    {
        if (p % 2 == 1)
        {
            r = r * k;
        }
        k = k *k;
        p = p / 2;
    }
    return r;
}
```

   a) Prouver la terminaison de cet algorithme.
   b) Prouver la correction de cet algorithme.
   ⊗ On pourra utiliser l'invariant de boucle suivant : $r \times k^p = a^n$.
   c) Donner la complexité de cette algorithme en fonction de l'exposant entier n.
   d) Prevoir l'affichage produit par la ligne de code suivante :
   `printf("2**32 = %u\n",exprap(2,32));`
   Expliquer.

❑ **Exercice 2** : *Second maximum*

1. Proposer un algorithme permettant de calculer le deuxième plus grand élément d'un tableau d'entiers. On suppose que le tableau contient toujours plus de deux éléments. Par exemple pour le tableau `[2, 10, 5, 17, 9]`, l'algorithme renvoie 10.
2. Prouver la correction de cet algorithme
3. Déterminer sa complexité. Est-il possible d'obtenir une complexité linéaire pour cet algorithme ?
4. Proposer une implémentation en langage C.
5. Proposer une implémentation impérative à l'aide du type `array` en OCaml.

❑ **Exercice 3** : *Liste chainées en C*

On défini le type maillon en C, comme un `struct` contenant un champ valeur et un champ pointeur vers le maillon suivant. Le type liste chainée est alors représentée par un pointeur vers un maillon :

```c
struct maillon
{
    int valeur;
    struct maillon *suivant;
};
typedef struct maillon maillon;
typedef maillon *liste;
```

On considère la fonction `maxliste` qui renvoie la plus grande valeur présente dans cette liste.

```c
int maxliste(liste l)
{
    int cmax = l->valeur;
    while (l != NULL)
    {
        if (l->valeur > cmax)
        {
            cmax = l->valeur;
        }
        l = l->suivant;
    }
    return cmax;
}
```

1. Que se passe-t-il si on appelle cette fonction avec une liste vide. Proposer une correction.
2. Prouver la terminaison de cette fonction.
3. Prouver la correction de cette fonction.
4. Ecrire une nouvelle fonction `supprime_max` qui supprime la première occurrence du maximum des éléments de la liste non vide donnée en paramètre.

❑ **Exercice 4** : *Calcul des termes de la suite de Fibonnacci*

On rappelle que la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ est définie par $F_0 = 1$, $F_1 = 1$ et $\forall n \geq 2, F_n = F_{n-1} + F_{n-2}$ (chaque terme est la somme des deux précédents)

1. Donner un algorithme récursif naïf permettant de calculer le nième terme de cette suite.
2. En proposer une implémentation en OCaml.
3. On note K_n le nombre d'appels récursifs nécessaires au calcul de F_n. Donner K_0, K_1 et la relation de récurrence vérifiée par $(K_n)_{(n\in\mathbb{N})}$
4. Etablir que $K_n = 2F_n - 1$.
5. En déduire la complexité de l'algorithme récursif naïf.
6. Donner un algorithme de complexité linéaire permettant de calculer F_n.
7. En fournir une implémentation en langage C.

❑ **Exercice 5** : *Suite « lock and say »*

La suite de Conway ou suite « *lock and say* » (regarder et dire) a pour premier terme $u_1 = 1$, puis chaque terme se détermine en énonçant les chiffres du terme précédent. Ainsi,
$u_2 = 11$ (car le terme précédent contient une fois le chiffre 1)
$u_3 = 21$ (car le terme précédent contient 2 fois le chiffre 1)
$u_4 = 1211$ (car le terme précédent contient une fois le chiffre 2 puis 1 fois le chiffre 1)

1. Déterminer les termes u_5, u_6, u_7 et u_8.
2. Proposer une conjecture sur les chiffres pouvant intervenir dans un terme de la suite.
3. Prouver cette conjecture
   ⊗ On pourra faire une preuve par récurrence et raisonner par l'absurde.
4. En utilisant le résultat établi à la question précédente, proposer une fonction en OCaml `int list -> int list` qui prend en argument un terme de la suite de Conway (sous la forme de la liste de ses chiffres) et renvoie le terme suivant (toujours sous la forme de la liste de ses chiffres)
   ⊗ Utiliser une correspondance de motif sur les chiffres par groupe de 3.
5. Ecrire une fonction en OCaml utilisant la fonction précédente et calculant le nième terme de la suite de Conway.

❑ **Exercice 6** : *Diviseurs et nombres parfaits*

1. Proposer un algorithme qui prend en entrée un entier nature n et renvoie la liste des diviseurs positif sde n. Par exemple pour l'entrée 10, l'algorithme renvoie la liste `[1, 2, 5, 10]`
2. Etudier la complexité de cet algorithme.

3. Prouver sa correction totale.
4. On dit qu'un nombre est *parfait* lorsqu'il est égal à la somme de ses diviseurs stricts. Proposer un algorithme qui prend en entrée un entier n et renvoie `true` si n est parfait et `false` sinon.
5. Quelle est la complexité de cet algorithme ?
6. Proposer une implémentation en OCaml des deux algorithmes.

❑ **Exercice 7** : *Suite de Golomb*

1. Donner un algorithme permettant de compter le nombre d'occurence d'un élément dans une liste.
2. Donner sa complexité.
3. En donner une implémentation dans le langage de votre choix (en C, on considère que la liste est une liste chainée d'entiers).
4. La suite de Golomb (du nom du mathématicien S. Golomb (1932–2016)), est la suite *croissante* d'entiers $(g_n)_{n\in\mathbb{N}^*}$ telle que $g_1 = 1$, $g_2 = 2$ et g_n est le nombre d'apparitions de n dans $(g_n)_{n\in\mathbb{N}^*}$.
   a) Donner les 20 premières termes de cette suite.
   b) Proposer un algorithme permettant de calculer le nième terme de cette suite.
   c) Etudier sa complexité.
   d) En proposer une implémentation dans le langage de votre choix.

❑ **Exercice 8** : *Calcul du* PGCD

1. Rappeler l'algorithme d'Euclide pour la calcul du PGCD de deux entiers naturels a et b.
2. Prouver la terminaison de cet algorithme.
3. Prouver la correction totale de cet algorithme.
4. Donner une implémentation récursive de cet algorithme en OCaml.
5. Donner une implémentation itérative de cet algorithme en C.

❑ **Exercice 9** : *Recherche dichotomique*

1. Rappeler le principe de l'algorithme de recherche par dichotomie dans un tableau. Préciser les préconditions.
2. Donner la complexité de cet algorithme.
3. Prouver sa terminaison.
4. Proposer une implémentation itérative en C.

❑ **Exercice 10** : *Représentation des entiers*

1. Ecrire $\overline{218}^{10}$ et $\overline{173}^{10}$ en base 2.
2. Ecrire $\overline{10101001}^{2}$ en base 10.
3. Donner l'écriture en complément à 2 sur 8 bits de $\overline{-100}^{10}$.
4. Proposer un algorithme permettant d'écrire un nombre donné en base 10 dans une autre base $b > 1$.
5. Prouver la terminaison de cet algorithme.
6. Etudier sa complexité.
7. En proposer une implémentation itérative en C.
8. En proposer une implémentation récursive en OCaml.

❑ **Exercice 11** : *Algorithme de tri*

1. Donner un algorithme de complexité quadratique.
2. Prouver sa correction totale.
3. En proposer une implémentation en langage C permettant de trier en place un tableau d'entiers.
4. Existe-t-il des algorithmes de tri ayant une meilleure complexité ?