❑ Exercice 1 : *Notation O*

1. Déterminer un O des suites de terme général :
   a) $2023n^2$ b) $n^2 + 10^9 n$ c) $3n + 7\log n$
   d) $2^{n+7} + n^{10}$ e) $\sqrt{19n^2 + 3}$ f) $\log(3n) + \log(n)$
2. Montrer que si $u_n = O(v_n)$ et $v_n = O(w_n)$ alors $u_n = O(w_n)$.
3. Montrer que $O(u_n + v_n) = O(\max(u_n, v_n))$.
4. Montrer que si $u_n = O(a_n)$ et $v_n = O(b_n)$ alors $u_n v_n = O(a_n b_n)$.
5. Déterminer un O (le « meilleur » possible) des expressions suivantes :
   a) $O(n^4) + O(n^2)$ b) $O(n^5) + O(n^5)$ b) $O(n^3) + O(\log(n))$
   d) $O(n^4) \times O(n^3)$ e) $O(n^4) \times O(\sqrt{n})$ f) $O(n^2) \times O(\log n)$

❑ Exercice 2 : *Vérification du tri*

1. Ecrire un algorithme permettant de vérifier qu'un tableau est trié par ordre croissant.
2. En proposer une implémentation en OCaml permettant de vérifier qu'une liste est triée.
3. Prouver que votre algorithme est correct.
4. Déterminer sa complexité.

❑ Exercice 3 : *multiplier en additionnant*

```c
int mult(int n, int p){
    int prod = 0;
    while (p>0){
        prod = prod + n;
        p = p -1;}
    return prod;}
```

1. En supposant $p > 0$ montrer la terminaison.
2. Prouver que cette fonction renvoie $p \times n$.
3. Déterminer sa complexité.

❑ Exercice 4 : *exponentiation rapide*

On rappelle la fonction d'exponentiation rapide dans sa version récursive :

```c
float expo(float a, int n){
    float cp = a;
    float res = 1;
    while (n!=0){
        if (n%2==1){
            res = res*cp;}
        cp = cp*cp;
        n=n/2;}
    return res;}
```

1. Prouver que cet algorithme termine.
2. Prouver qu'il est correct.
   En notant n_0 la valeur initiale de n, on pourra considérer l'invariant suivant : `res * cp`$^{\texttt{n}}$$=a^{n_0}$
3. Donner sa complexité.

❑ Exercice 5 : *retour sur la multiplication*

On donne la fonction suivante :

```c
int multiplie(int n, int p){
    int prod = 0;
    while (n>0){
        if (n%2==1) {
            prod = prod+p;}
        n= n / 2;
        p = p*2;}
    return prod;}
```

1. Vérifier à la main, sur deux entiers naturels de votre choix que cette fonction est conforme à sa spécification.
2. Montrer la terminaison de cette fonction
3. Montrer que cette fonction est bien conforme à sa spécification.

❏ Exercice 6 : *tri à bulles*

1. Rappeler le principe du tri à bulles.
2. En écrire une implémentation en C, dans laquelle on vérifie à chaque passage qu'au moins une inversion a été effectuée. Si tel n'est pas le cas on termine immédiatement l'algorithme puisque cela signifie que les élements sont triées.
3. Montrer la terminaison de cette fonction
4. Prouver qu'elle est correcte.
   On pourra exhiber un invariant qui donne le nombre d'éléments figurants à leur place finale après chaque itération.

❏ Exercice 7 : *majorité absolue*

On considère les résultats d'un vote sous la forme d'un tableau d'entier positifs, lorsqu'on rencontre la valeur `i` cela signifie que la candidat numéro `i` a obtenu un vote. Par exemple si le tableau contient les valeurs `[2, 3, 1, 2, 2, 2, 4, 1, 1, 2]`, alors le candidat 1 a obtenu 3 voix, le candidat 2 a obtenu 5 voix, ... On cherche à déterminer un algorithme efficace permettant de déterminer (s'il existe) le candidat ayant obtenu la majorité absolue. Dans le cas où aucun candidat n'a la majorité absolue alors l'algorithme doit renvoyer `-1`. On note C le nombre de candidats et N le nombre de votes.

1. Proposer un algorithme permettant de résoudre ce problème et donner sa complexité
   On donne ci-dessous **le début** de l'implémentation en C d'un algorithme proposé par R. Boyer et S. Moore :

```c
int majorite_absolue(int vote[], int size){
    int nb_votes = 0;
    int candidat = 0;
    for (int i = 0; i < size; i++){
        if (nb_votes == 0){
            candidat = vote[i];
            nb_votes = 1;}
        else{
            if (vote[i] == candidat){
                nb_votes += 1;}
            else{
                nb_votes -= 1;}
        }
    }
```

2. Prouver qu'à la fin de la boucle `for`, la variable `candidat` contient le numéro du seul candidat éventuellement majoritaire
3. Compléter l'implémentation en ajoutant les lignes permettant de vérifier après la boucle for que ce candidat est effectivment majoritaire
4. Donner la complexité de cet algorithme.

❑ **Exercice 8** : *nombre de chiffres d'un entier*

1. Ecrire en C, une version itérative d'une fonction donnant le nombre de chiffres d'un entier naturel.
2. Ecrire une version récursive en OCaml.
3. Prouver la terminaison dans les deux cas.
4. Prouver la correction dans les deux cas.