Exercice 1 : La notation For

1. En utilisant une compréhension **for**

```
def divisors(xs: List[Int]): List[(Int, Int)] =
  for (x <- xs; y <- xs if y % x == 0 && x != y) yield (x, y)
```

2. En utilisant les méthodes *filter*, *map* et *flatMap*

```
def divisors(xs: List[Int]): List[(Int, Int)] =
 xs.flatMap(x =>
  xs.filter(y => y != x && y % x == 0).map((x, _)))
```

Exercice 2 : Filtrage de motifs et récursion

Partie 1 Run-Length Encoding

```
def encode(xs: List[Char]): List[(Char, Int)] = xs match {
    case x :: y :: ys if x == y =>
      val (z, n) :: zs = encode(y :: ys)
      (z, n+1) :: zs
    case x :: y :: ys => (x, 1) :: encode(y :: ys)
    case x :: Nil => (x, 1) :: Nil
    case Nil => Nil
}
```

Partie 2 Maxima Locaux

```
def findMaxima(in: List[Int]): List[Int] = {
    def findMaxima0(xs: List[Int]): List[Int] = xs match {
      case x :: y :: z :: rest if (y - x > 0 && z - y < 0) =>
        y :: findMaxima0(y :: z :: rest)
      case x :: y :: z :: rest =>
        findMaxima0(y :: z :: rest)
      case x :: y :: Nil if (y - x > 0) => y :: Nil
      case other => Nil
    }

    in match {
      case x :: y :: rest if (y - x < 0) =>
        x :: findMaxima0(y :: rest)
      case other => findMaxima0(in)
    }
}
```

Exercice 3 : Tries

Partie 1

```
  def emptyTrie = Trie(false, Nil)
```

Partie 2

Récursivement parcourir le trie jusqu'à ce qu'il n'y pas plus de caractères. Si le nœud est terminal, le mot est dans le trie.

```
def parseWord(dict: Trie, word: Word): Boolean = word match {
    case x :: xs => parseWord(dict.next(x), xs)
    case Nil => dict.isTerminal
}
```

Partie 3

```
def parse(dict: Trie, sentence: Sentence): Boolean = sentence match {
    case x :: xs =>
      parseWord(dict, x) && parse(dict, xs)
    case Nil =>
      true
}
```

Partie 4

En règle générale, de mettre à jour un arbre immuable, nous modifions l'ascendant. Nous modifions le subtrie. Deuxièmement, nous modifions le nœud actuel avec la version actualisée de la subtrie.

```
  def add(trie: Trie, word: Word): Trie = word match {
    case x :: xs =>
      // Obtenez le subtrie pour le caractere suivant.
      // S'il n'ya pas de subtrie pour ce caractere,
      // `next` va construire un trie vide.
      val subtrie = trie.next(x)
      // Modifiez le subtrie recursivement avec le reste des caracteres.
      // Notez que si le subtrie est vide, ce sont toujours correctes.
      val updatedSubtrie = add(subtrie, xs)
      // Construire un nouveau noeud de celle-ci
      // avec le subtrie pour ce caractere mis à jour.
      trie.updateChild(x, updatedSubtrie)
    case Nil =>
      // S'il n'ya pas plus de caracteres dans le mot,
      // ce noeud doit être terminale.
      // Le subtries doivent rester les memes.
      Trie(true, trie.children)
  }
```

Partie 5

Nous ajoutons un mot à la trie, puis ajouter le reste des mots récursivement.

```
def addAll(trie: Trie, words: List[Word]): Trie = words match {
    case x :: xs => addAll(add(trie, x), xs)
    case Nil => trie
}
```