Exercice 1 : Manipulation de listes (15 points)

Partie 1

```
def prefixes[A](xs:List[A]):List[List[A]] = xs match {
  case List() => List(Nil)
  case y::ys  => List() :: (prefixes(ys) map { x => y :: x })
}
```

Partie 2

```
def suffixes[A](xs:List[A]):List[List[A]] = xs match {
  case List() => List(Nil)
  case y::ys  => xs :: suffixes(ys)
}
```

Exercice 2 : Preuve inductive (15 points)

```
case Nil
  rev(Nil:::ys)
  = app1
  rev(ys)

  rev(ys):::rev(Nil)
  = rev1
  rev(ys):::Nil
  = neutre
  rev(ys)
```

```
case z::zs
  rev((z::zs):::ys)
  = app2
  rev(z::(zs:::ys))
  = rev2
  rev(zs:::ys):::List(z)
  = hyp
  (rev(ys):::rev(zs)):::List(z)

  rev(ys):::rev(z::zs)
  = rev2
  rev(ys):::(rev(zs):::List(z))
  = assoc
  (rev(ys):::rev(zs)):::List(z)
```

Exercice 3 : Ni trop cher, ni trop calorique (15 points)

Partie 1

```
def genMenus(food:List[FoodItem],maxCal:Int,maxPrice:Double) =
  for(val s <- food; s.kind == Starter;
      val m <- food; m.kind == MainCourse;
      val d <- food; d.kind == Dessert;
      s.price+m.price+d.price < maxPrice;
      s.calories+m.calories+d.calories < maxCal)
    yield Triple(s,m,d)
```

Partie 2

```
  food.filter { s => s.kind == Starter }.flatMap { s =>
    food.filter { m => m.kind == MainCourse }.flatMap { m =>
      food.filter { d => d.kind == Dessert }
      .filter { d => s.price+m.price+d.price < maxPrice }
      .filter { d => s.calories+m.calories+d.calories < maxCalories)
        .map {
              Triple(s,m,d)
} } }
```

Exercice 4 : Queues fonctionnelles (25 points)

Partie 1

```
La complexité de ce program et O(N*N), parce que
* 'append' est O(N) et on l'appele N fois, ce qui vaut O(N*N)
* 'head' et 'tail' sont O(1), les appeler N est d'ordre O(N)
```

Partie 2

```
def head[A](q: Queue[A]): A = q match {
  case Pair(Nil, ys)    => ys.reverse.head
  case Pair(x :: xs, _) => x
}
def tail[A](q: Queue[A]): Queue[A] = q match {
  case Pair(Nil, ys)     => Pair(ys.reverse.tail, Nil)
  case Pair(x :: xs, ys) => Pair(xs, ys)
}
```

Partie 3

```
O(N), parce que
* 'append' est O(1) et on l'appele N fois, ce qui vaut O(N)
* 'reverse' est O(N)
* un (seul) des appels de 'head', 'tail' aura la complexité O(N)
 a cause du 'reverse'
* les autre N-1 appels de 'head' et de 'tail' seront O(1),
  alors encore O(N)
```