Fiche mémo Scala

```
scala> def sumInt(min:Int,max:Int): Int={
    |     if (min == max)
    |         min
    |     else
    |         min+sumInt(min+1,max)
    | }
sumInt: (min: Int, max: Int)Int

scala> def sumSquareInt(min:Int,max:Int): Int={
    |     if (min == max)
    |         min
    |     else
    |         min*min+sumInt(min+1,max)
    | }
sumSquareInt: (min: Int, max: Int)Int

scala> def sum(min:Int,max:Int,f:Int => Int):Int={
    |     if (min>max)
    |         0
    |     else
    |         f(min)+sum(min+1,max,f)
    | }
sum: (min: Int, max: Int, f: Int => Int)Int

scala> def squareSum(acc:Int,e:Int):Int={
    |     acc+e*e
    | }
squareSum: (acc: Int, e: Int)Int

scala> def elements(min:Int,max:Int):List[Int]={
    | if (min>max) Nil
    | else min :: elements(min+1,max)
    | }
elements: (min: Int, max: Int)List[Int]

scala> def foldLeft[A,B](l:List[A],f:(B,A)=>B,acc:B):B={
    | if (l.isEmpty) acc
    | else foldLeft(l.tail,f,f(acc,l.head))
    | }
foldLeft: [A, B](l: List[A], f: (B, A) => B, acc: B)B

scala> def fact(x:Int):Int={
    | if (x==0) 1
    | else x*fact(x-1)
    | }
fact: (x: Int)Int

scala> def factAcc0(x:Int, acc:Int):Int={
    | if (x==0) acc
    | else factAcc0(x-1,acc*x)
    | }
factAcc0: (x: Int, acc: Int)Int

scala> def factAcc(x:Int):Int=factAcc0(x,1)
factAcc: (x: Int)Int

scala> def generique(x:Int, acc:Int, f:(Int,Int)=>Int):Int={
    | if (x==0) acc
    | else generique(x-1,f(acc,x),f)
    | }
generique: (x: Int, acc: Int, f: (Int, Int) => Int)Int

scala> import scala.annotation.tailrec

scala> def facto(value: BigInt): BigInt = {
    | require(value>0, s"first paramater should no be negative :
$value")
    | @tailrec
    | def factRT(value: BigInt, acc: BigInt): BigInt = {
    | if (value < 2) acc
    | else factRT(value-1, acc * value)
    | }
    | factRT(value, 1)
    | }
facto: (value: BigInt)BigInt

scala> def compose[A,B,C](f:B=>C,g:A=>B):A=>C={x => f(g(x))}
compose: [A, B, C](f: B => C, g: A => B)A => C

scala> def inv(x:Int):Double=1.0/x
inv: (x: Int)Double

scala> def squarei(x:Double):Double=x*x
squarei: (x: Double)Double

scala> def addi(x:Double,y:Double):Double=x+y
addi: (x: Double, y: Double)Double

scala> def
sum[B,C](min:Int,max:Int,f:Int=>B,add:(C,B)=>C,acc:C):C={
    | if (min>max) acc
    | else sum(min+1,max,f,add,add(acc,f(min)))
    | }
sum: [B, C](min: Int, max: Int, f: Int => B, add: (C, B) => C, acc:
C)C

scala> sum(1,2,compose(squarei,inv),addi,0.0)
res0: Double = 1.25

scala> import scala.annotation.tailrec

scala> @tailrec
    | def last[A](l:List[A]):A=l match {
    | case h :: Nil => h
    | case _ :: q => last(q)
    | case _ => error(s"List Nil has no last")
    | }
warning: there was one deprecation warning; re-run with
-deprecation for details
last: [A](l: List[A])A

scala> @tailrec
    | def anteLast[A](l:List[A]):A= l match {
    | case h :: _ :: Nil => h
    | case _ :: h :: q => anteLast(h::q)
    | case _ => error(s"List has no ante-last")
    | }
warning: there was one deprecation warning; re-run with
-deprecation for details
anteLast: [A](l: List[A])A

scala> @tailrec
    | def nth[A](n:Int,l:List[A]):A= (n,l) match {
    | case (0,h :: _) => h
    | case (_,_ :: q) if n>0 => nth(n-1,q)
    | case _ => error(s"List has no nth element")
    | }
warning: there was one deprecation warning; re-run with
-deprecation for details
nth: [A](n: Int, l: List[A])A

scala> def length[A](l:List[A]):Int= l match{
    | case Nil => 0
    | case _ :: l2 => 1+length(l2)
    | }
length: [A](l: List[A])Int

scala> @tailrec
    | def length_aux[A](l:List[A],acc:Int):Int= l match{
    | case Nil => acc
    | case _ :: l2 => length_aux(l2,acc+1)
    | }
```

Fiche mémo Scala

```
length_aux: [A](l: List[A], acc: Int)Int

scala> def length[A](l:List[A]):Int=length_aux(l,0)
length: [A](l: List[A])Int

scala> @tailrec
    | def reverse_aux[A](l:List[A],acc:List[A]):List[A]=l match {
    | case Nil => acc
    | case h :: t => reverse_aux(t,h::acc)
    | }
reverse_aux: [A](l: List[A], acc: List[A])List[A]

scala> def reverse[A](l:List[A]):List[A]=reverse_aux(l,List())
reverse: [A](l: List[A])List[A]

scala> def isPalindrome[A](l:List[A]):Boolean=l==reverse(l)
isPalindrome: [A](l: List[A])Boolean

scala> def duplicate[A](a:Int,l:List[A],acc:List[A]):List[A]=a match{
    | case 0 => acc
    | case _ => duplicate(a-1,l,l:::acc)
    | }
duplicate: [A](a: Int, l: List[A], acc: List[A])List[A]

scala> def flatten[A](l:List[Any]):List[Any]= l match {
    | case List(0,t:Any) => List()
    | case List(a:Int,t:List[Any]) => duplicate(a,flatten(t),List())
    | case n => List(n)
    | }
flatten: [A](l: List[Any])List[Any]

scala> def filter[A](l:List[A],f:A=>Boolean):List[A]=l match{
    | case Nil => Nil
    | case h :: t if f(h) => h::filter(t,f)
    | case _ :: t => filter(t,f)
    | }
filter: [A](l: List[A], f: A => Boolean)List[A]

scala> def bar (x:Int):Int={if (x>=5) x else 0}
bar: (x: Int)Int

scala> l.map(bar)
<console>:14: error: not found: value l

scala> val l= List(1, 4, 8, 5, 6, 2, 3, 4)
l: List[Int] = List(1, 4, 8, 5, 6, 2, 3, 4)

scala> val l = Range(1,6).toList
l: List[Int] = List(1, 2, 3, 4, 5)

scala> def bar (x:Int,y:Int)=x+y
bar: (x: Int, y: Int)Int

scala> def bar (x:Int,y:Int)=x-y
bar: (x: Int, y: Int)Int

scala> def bar(x:String,y:Any):String=x+"->"+y.toString
bar: (x: String, y: Any)String

scala> l.foldLeft("List:")(bar)
res2: String = List:->1->2->3->4->5

scala> def bar(y:Any,x:String):String=x+"->"+y.toString
bar: (y: Any, x: String)String

scala> l.foldRight("List:")(bar)
res3: String = List:->5->4->3->2->1

scala> def bar (x:Int):Boolean={x>5}
bar: (x: Int)Boolean

scala> def filter[A](l:List[A],f:A=>Boolean):List[A]={
    |     def foo(f:A=>Boolean)(acc:List[A],e:A):List[A]={
    |     if (f(e)) acc:::List(e)
    |     else acc
    |     }
    |     val acc:List[A]=List()
    |     l.foldLeft(acc)(foo(f))
    | }
filter: [A](l: List[A], f: A => Boolean)List[A]

scala> def bar (x:Int):Int={if (x>=5) x else 0}
bar: (x: Int)Int

scala> def map[A,B](l:List[A],f:A=>B):List[B]={
    |     def foo(f:A=>B)(acc:List[B],e:A):List[B]={
    |     acc:::List(f(e))
    |     }
    |     val acc:List[B]=List()
    |     l.foldLeft(acc)(foo(f))
    | }
map: [A, B](l: List[A], f: A => B)List[B]

scala> def foo[A](l:List[A],f:A=>Boolean,v:A):List[A]={
    |     def bar[A](f:A=>Boolean,v:A)(a:A):A={
    |         if (f(a)) a
    |         else v
    |         }
    |     l.map(bar(f,v))
    | }
foo: [A](l: List[A], f: A => Boolean, v: A)List[A]

scala> def bar(a:Int):Boolean=a%2==0
bar: (a: Int)Boolean

scala> foo(List(1,5,8,6,4,8,9,2,1,12),bar,0)
res4: List[Int] = List(0, 0, 8, 6, 4, 8, 0, 2, 0, 12)

scala> def bar(a:String):Boolean=a.startsWith("a")
bar: (a: String)Boolean

scala>
foo(List("adgs","azfdwf","srg","azddwxfd","sfdgsg","sgfds"),bar,"")
res5: List[String] = List(adgs, azfdwf, "", azddwxfd, "", "")

scala> import scala.annotation.tailrec
import scala.annotation.tailrec

scala> case class Rational (p0 : Int, q0 : Int=1){
    |     require(q0!= 0, "denominator must be nonzero")
    |     private val gcd = {
    |         @tailrec
    |         def gcdRec(x: Int, y: Int): Int = {
    |             if(y == 0) x
    |             else gcdRec(y, x % y)
    |         }
    |         gcdRec(p0, q0)
    |     }
    |     val p=p0/gcd
    |     val q=q0/gcd
    |
    |     def unary_- = new Rational(-p, q)
    |
    |     def +(that: Rational) =
    |         Rational(that.p * q + p * that.q, q * that.q)
    |
    |     def -(that: Rational) = this + -that
    |
    |     def *(that: Rational) = Rational(p * that.p, q * that.q)
    |
    |     def /(that: Rational) = Rational(p * that.q, q * that.p)
    |
```

```
     |     def <(that: Rational) = p * that.q < that.p * q
     |
     |     def >(that: Rational) = !(this < that)
     |
     |     def max(that: Rational) = if (this < that) that else this
     |
     |     def ==(that: Rational) = {that.p == p && that.q == q}
     |
     |     override def toString =p+"/"+q
     | }
defined class Rational

scala> new Rational(4,8)
res6: Rational = 1/2

scala> case class
NormalHuman(firstName:String,lastName:String,age:Option[Int])
defined class NormalHuman

scala> case class
SuperHero(nickName:String,humain:Option[NormalHuman],super
Power:Any)
defined class SuperHero

scala>
SuperHero("Superman",Some(NormalHuman("Clark","Kent",Som
e(35))),"OverPower")
res7: SuperHero =
SuperHero(Superman,Some(NormalHuman(Clark,Kent,Some(35)
)),OverPower)

scala>
SuperHero("Superman",Some(NormalHuman("Clark","Kent",None
)),"OverPower")
res8: SuperHero =
SuperHero(Superman,Some(NormalHuman(Clark,Kent,None)),Ov
erPower)

scala> SuperHero("Superman",None,List(1,2,3))
res9: SuperHero = SuperHero(Superman,None,List(1, 2, 3))

scala> NormalHuman("Martin","Pozuelo",Some(21))
res10: NormalHuman = NormalHuman(Martin,Pozuelo,Some(21))

scala> NormalHuman("Martin","Pozuelo",None)
res11: NormalHuman = NormalHuman(Martin,Pozuelo,None)

scala> for (i <- 0 until 10) {
     | println(i)
     | }
0
1
2
3
4
5
6
7
8
9

scala> for (i:Int <- 0 until 10) {
     | println(i)
     | }
0
1
2
3
4
5
6
7
8
9

scala> Range(0,10).foreach(println)
0
1
2
3
4
5
6
7
8
9

scala> val l1 = for (i <- 0 until 10) yield i*2
l1: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 2, 4, 6,
8, 10, 12, 14, 16, 18)

scala> val l2 = Vector.range(0,10).map(_*2)
l2: scala.collection.immutable.Vector[Int] = Vector(0, 2, 4, 6, 8, 10,
12, 14, 16, 18)

scala> val l1 = for {i <- 0 until 10
     | j <- 0 until 10} yield Vector(i,j)
l1:
scala.collection.immutable.IndexedSeq[scala.collection.immutable.
Vector[Int]] = Vector(Vector(0, 0), Vector(0, 1), Vector(0, 2),
Vector(0, 3), Vector(0, 4), Vector(0, 5), Vector(0, 6), Vector(0, 7),
Vector(0, 8), Vector(0, 9), Vector(1, 0), Vector(1, 1), Vector(1, 2),
Vector(1, 3), Vector(1, 4), Vector(1, 5), Vector(1, 6), Vector(1, 7),
Vector(1, 8), Vector(1, 9), Vector(2, 0), Vector(2, 1), Vector(2, 2),
Vector(2, 3), Vector(2, 4), Vector(2, 5), Vector(2, 6), Vector(2, 7),
Vector(2, 8), Vector(2, 9), Vector(3, 0), Vector(3, 1), Vector(3, 2),
Vector(3, 3), Vector(3, 4), Vector(3, 5), Vector(3, 6), Vector(3, 7),
Vector(3, 8), Vector(3, 9), Vector(4, 0), Vector(4, 1), Vector(4, 2),
Vector(4, 3), Vector(4, 4), Vector(4, 5), Vector(4, 6), Vector(4, 7),
Vector(4, 8), Vector(4, 9), Vector...
scala> val l2 = Vector.range(0,10).flatMap{ x =>
Vector.range(0,10).map{y=> Vector(x,y)}}
l2:
scala.collection.immutable.Vector[scala.collection.immutable.Vecto
r[Int]] = Vector(Vector(0, 0), Vector(0, 1), Vector(0, 2), Vector(0, 3),
Vector(0, 4), Vector(0, 5), Vector(0, 6), Vector(0, 7), Vector(0, 8),
Vector(0, 9), Vector(1, 0), Vector(1, 1), Vector(1, 2), Vector(1, 3),
Vector(1, 4), Vector(1, 5), Vector(1, 6), Vector(1, 7), Vector(1, 8),
Vector(1, 9), Vector(2, 0), Vector(2, 1), Vector(2, 2), Vector(2, 3),
Vector(2, 4), Vector(2, 5), Vector(2, 6), Vector(2, 7), Vector(2, 8),
Vector(2, 9), Vector(3, 0), Vector(3, 1), Vector(3, 2), Vector(3, 3),
Vector(3, 4), Vector(3, 5), Vector(3, 6), Vector(3, 7), Vector(3, 8),
Vector(3, 9), Vector(4, 0), Vector(4, 1), Vector(4, 2), Vector(4, 3),
Vector(4, 4), Vector(4, 5), Vector(4, 6), Vector(4, 7), Vector(4, 8),
Vector(4, 9), Vector(5, ...

scala> val v1 = for {i <- 0 until 10 if (i%2 == 0)} yield i
v1: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 2, 4, 6,
8)

scala> import scala.util.Random
import scala.util.Random

scala> val o1 = if (Random.nextInt % 2 == 0)
Some(Random.nextInt) else None
o1: Option[Int] = None

scala> val o2 = if (Random.nextInt % 2 == 0)
Some(Random.nextInt) else None
o2: Option[Int] = None

scala> val o3 = for {
```

```
    | v1 <- o1
    | v2 <- o2
    | } yield (v1 + v2)
o3: Option[Int] = None

scala> def produceAnOption = if (Random.nextInt % 2 == 0)
Some(Random.nextInt) else None
produceAnOption: Option[Int]

scala> val v1 = Vector.fill(10)(produceAnOption)
v1: scala.collection.immutable.Vector[Option[Int]] =
Vector(Some(599164262), None, None, Some(257572459), None,
None, None, Some(-226968580), Some(-1548675542), None)

scala> val v2 = Vector.fill(10)(produceAnOption)
v2: scala.collection.immutable.Vector[Option[Int]] = Vector(None,
Some(340925477), None, None, Some(447728825),
Some(-776820581), Some(-1786776745), Some(-340216815),
Some(1184115154), Some(68090504))

scala> for {
    | optionI <- v1
    | optionJ <- v2
    | i <- optionI
    | j <- optionJ
    | } yield (i+j) // You will have a vector of Int !
res15: scala.collection.immutable.Vector[Int] = Vector(940089739,
1046893087, -177656319, -1187612483, 258947447,
1783279416, 667254766, 598497936, 705301284, -519248122,
-1529204286, -82644356, 1441687613, 325662963, 113956897,
220760245, -1003789161, -2013745325, -567185395, 957146574,
-158878076, -1207750065, -1100946717, 1969471173,
959515009, -1888892357, -364560388, -1480585038)

scala> def mergeSort [A](l : List[A])(implicit ord: Ordering[A]) :
List[A] = {
    |     def merge(l1: List[A], l2: List[A]): List[A] =(l1, l2) match {
    |         case (Nil, _) => l2
    |         case (_, Nil) => l1
    |         case (h1::t1,h2::t2) =>
    |             if (ord.lteq(h1,h2))  h1 :: merge(t1,l2)
    |             else h2 :: merge(l1,t2)
    |     }
    |     val milieu = l.length/2
    |     if (milieu == 0) l
    |     else {
    |         val (l1,l2) = l splitAt milieu
    |         merge(mergeSort(l1),mergeSort(l2))
    |     }
    | }
mergeSort: [A](l: List[A])(implicit ord: Ordering[A])List[A]

scala> val lInt =
List.fill(10)(math.random).map(_*1000).map(_.toInt)
lInt: List[Int] = List(209, 863, 160, 864, 771, 806, 983, 986, 600,
963)

scala> mergeSort(lInt) // works
res16: List[Int] = List(160, 209, 600, 771, 806, 863, 864, 963, 983,
986)

scala> val lDouble = List.fill(10)(math.random).map(_*100)
lDouble: List[Double] = List(59.1693031727141,
19.266535673848463, 52.400566614746346,
3.502037069064501, 75.89633969650662, 54.77321934479493,
35.57830317978766, 94.05560431933144, 30.52586634782457,
36.030053178960294)

scala> mergeSort(lDouble) // works now
res17: List[Double] = List(3.502037069064501,
19.266535673848463, 30.52586634782457, 35.57830317978766,
36.030053178960294, 52.400566614746346,
54.77321934479493, 59.1693031727141, 75.89633969650662,
94.05560431933144)

scala> def quickSort [A](l : List[A])(implicit ord: Ordering[A]) :
List[A] = {
    |     if (l == List()) l
    |     else
quickSort(l.tail.filter(ord.gt(l.head,_))):::(l.head::quickSort(l.tail.filter(
ord.lteq(l.head,_))))
    | }
quickSort: [A](l: List[A])(implicit ord: Ordering[A])List[A]

scala> val lInt =
List.fill(10)(math.random).map(_*1000).map(_.toInt)
lInt: List[Int] = List(557, 442, 358, 347, 226, 430, 827, 388, 370,
284)

scala> quickSort(lInt) // works
res18: List[Int] = List(226, 284, 347, 358, 370, 388, 430, 442, 557,
827)

scala> val lDouble = List.fill(10)(math.random).map(_*100)
lDouble: List[Double] = List(56.67123931007563,
96.59178184484693, 80.48739402658579, 73.07318705692774,
86.64927224362715, 75.14103194204678, 16.719103066178633,
59.52608157621171, 32.088387382762484, 71.54545098560143)

scala> quickSort(lDouble) // works now
res19: List[Double] = List(16.719103066178633,
32.088387382762484, 56.67123931007563, 59.52608157621171,
71.54545098560143, 73.07318705692774, 75.14103194204678,
80.48739402658579, 86.64927224362715, 96.59178184484693)

scala> def toInt2 (s : String):Any={
    | try {
    |     s.toInt
    |   } catch {
    |     case e: Exception => None
    |   }
    | }
toInt2: (s: String)Any

scala> var l = List("14","a","45","ae")
l: List[String] = List(14, a, 45, ae)

scala> l.map(toInt2)
res20: List[Any] = List(14, None, 45, None)

scala> def f(x: Int):Option[Int] = if (x > 2) Some(x) else None
f: (x: Int)Option[Int]

scala> def f(x: Int)= if (x > 2) Some(x) else None
f: (x: Int)Option[Int]

scala> val l = List(1,2,3,4,5)
l: List[Int] = List(1, 2, 3, 4, 5)

scala> l.map(x => f(x))
res21: List[Option[Int]] = List(None, None, Some(3), Some(4),
Some(5))

scala> def g(v:Int) = List(v-1, v, v+1)
g: (v: Int)List[Int]

scala> l.map(x => g(x))
res22: List[List[Int]] = List(List(0, 1, 2), List(1, 2, 3), List(2, 3, 4),
List(3, 4, 5), List(4, 5, 6))

scala> l.flatMap(x => g(x))
res23: List[Int] = List(0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 6)
```

Fiche mémo Scala

```
scala> def isPrime(e:Int):Boolean={
     |     (for (i <- 1 to e if e%i==0) yield e).length==2
     | }
isPrime: (e: Int)Boolean

scala> for {i <- 0 to 10
     | j <- 0 to 10 if isPrime(i+j) } yield (i,j)
res24: scala.collection.immutable.IndexedSeq[(Int, Int)] =
Vector((0,2), (0,3), (0,5), (0,7), (1,1), (1,2), (1,4), (1,6), (1,10), (2,0),
(2,1), (2,3), (2,5), (2,9), (3,0), (3,2), (3,4), (3,8), (3,10), (4,1), (4,3),
(4,7), (4,9), (5,0), (5,2), (5,6), (5,8), (6,1), (6,5), (6,7), (7,0), (7,4),
(7,6), (7,10), (8,3), (8,5), (8,9), (9,2), (9,4), (9,8), (9,10), (10,1),
(10,3), (10,7), (10,9))

scala> var n=50
n: Int = 50

scala> for {
     | a <- 0 to n
     | b <- a+1 to n
     | c <- b+1 to n
     | if a*a+b*b==c*c
     | } yield (a,b,c)
res25: scala.collection.immutable.IndexedSeq[(Int, Int, Int)] =
Vector((3,4,5), (5,12,13), (6,8,10), (7,24,25), (8,15,17), (9,12,15),
(9,40,41), (10,24,26), (12,16,20), (12,35,37), (14,48,50),
(15,20,25), (15,36,39), (16,30,34), (18,24,30), (20,21,29),
(21,28,35), (24,32,40), (27,36,45), (30,40,50))

scala> class Memoize1[-T, +R](f: T => R) extends (T => R) {
     |   import scala.collection.mutable
     |   private[this] val vals = mutable.Map.empty[T, R]
     |
     |   def apply(x: T): R = {
     |     if (vals.contains(x)) {
     |       vals(x)
     |     }
     |     else {
     |       val y = f(x)
     |       vals += ((x, y))
     |       y
     |     }
     |   }
     | }
defined class Memoize1

scala> def fibo(n:Int):Int={
     |     if (n<2) 1 else fibo(n-1)+fibo(n-2)
     | }
fibo: (n: Int)Int

scala> object FiboMemoization{
     |     val dict=scala.collection.mutable.Map[BigInt,BigInt]()
     |
     |     def apply(n:BigInt):BigInt={
     |         dict.getOrElseUpdate(n,if (n<3) 1 else
apply(n-1)+apply(n-2))
     |         dict(n)
     |     }
     | }
defined object FiboMemoization

scala> var file : String = "dico"
file: String = dico

scala> type Word = String
defined type alias Word

scala> type Sentence = List[Word]
defined type alias Sentence

scala> val lines = scala.io.Source.fromFile(file).mkString
lines: String =
"aast
abainville
abancourt
abancourt
abaucourt hautecourt
abaucourt sur seille
abbans dessous
abbans dessus
abbaretz
abbazia
abbecourt
abbecourt
abbenans
abbeville
abbeville
abbeville la riviere
abbeville les conflans
abbeville st lucien
abbevillers
abeilhan
abelcourt
abere
abergement la ronce
abergement le grand
abergement le petit
abergement les thesy
abergement st jean
aberwrach
abidos
abilly
abitain
abjat sur bandiat
ablain st nazaire
ablaincourt pressoir
ablainzevelle
ablancourt
ableiges
abl...

scala> type Occurrences = List[(Char, Int)]
defined type alias Occurrences

scala> "test".groupBy(identity).mapValues(_.size).toList
res26: List[(Char, Int)] = List((e,1), (t,2), (s,1))

scala> val dictionary: List[Word] = lines.split("\n").toList
dictionary: List[Word] = List(aast, abainville, abancourt, abancourt,
abaucourt hautecourt, abaucourt sur seille, abbans dessous,
abbans dessus, abbaretz, abbazia, abbecourt, abbecourt,
abbenans, abbeville, abbeville, abbeville la riviere, abbeville les
conflans, abbeville st lucien, abbevillers, abeilhan, abelcourt,
abere, abergement la ronce, abergement le grand, abergement le
petit, abergement les thesy, abergement st jean, aberwrach,
abidos, abilly, abitain, abjat sur bandiat, ablain st nazaire,
ablaincourt pressoir, ablainzevelle, ablancourt, ableiges, ablis,
ablon, ablon sur seine, aboen, aboncourt, aboncourt, aboncourt
gesincourt, aboncourt sur seille, abondance, abondant, abos,
abreschviller, abrest, abries, abscon, abzac, abzac, accolans,
accolay, accons, accous, achain, achen,...

scala> def wordOccurrences(w: Word): Occurrences =
w.toLowerCase.groupBy(identity).mapValues(_.length).toList.sortB
y(identity)
wordOccurrences: (w: Word)Occurrences

scala> def sentenceOccurrences(s: Sentence): Occurrences =
s.flatMap(wordOccurrences).groupBy(_._1).mapValues((_.map(_._
2).sum)).toList.sortBy(identity)
```

Fiche mémo Scala

```
sentenceOccurrences: (s: Sentence)Occurrences

sentenceOccurrences(List("azsqaz","aqzzs","qazqqaszqa"))
res27: Occurrences = List((a,6), (q,6), (s,3), (z,6))

scala> def sentenceOccurrences2(s: Sentence): Occurrences =
wordOccurrences(s.mkString(""))
sentenceOccurrences2: (s: Sentence)Occurrences

scala>
sentenceOccurrences(List("azsqaz","aqzzs","qazqqaszqa"))
res28: Occurrences = List((a,6), (q,6), (s,3), (z,6))

scala> lazy  val dictionaryByOccurrences: Map[Occurrences,
List[Word]] = dictionary.groupBy(wordOccurrences)
dictionaryByOccurrences: Map[Occurrences,List[Word]] = <lazy>

scala> def wordAnagrams(word: Word): List[Word] =
dictionaryByOccurrences(wordOccurrences(word))
wordAnagrams: (word: Word)List[Word]

scala> wordAnagrams("gornac")
res29: List[Word] = List(gornac, rognac)

scala> wordAnagrams("paris la défense")
res30: List[Word] = List(paris la défense, paris la défense, paris la
défense, paris la défense, paris la défense, paris la défense, paris
la défense, paris la défense, paris la défense, paris la défense,
paris la défense, paris la défense, paris la défense, paris la
défense, paris la défense, paris la défense, paris la défense, paris
la défense, paris la défense, paris la défense, paris la défense,
paris la défense, paris la défense, paris la défense, paris la
défense, paris la défense, paris la défense, paris la défense, paris
la défense...

dictionaryByOccurrences.map(x=>x._2).maxBy(_.length)
res31: List[Word] = List(paris la défense, paris la défense, paris la
défense, paris la défense, paris la défense, paris la défense, paris
la défense, paris la défense, paris la défense, paris la défense,
paris la défense, paris la défense, paris la défense, paris la
défense, paris la défense, paris la défense, paris la défense, paris
la défense, paris la défense, paris la défense, paris la défense,
paris la défense, paris la défense, paris la défense, paris la
défense, paris la défense, paris la défense, paris la défense, paris
la défense...

scala> val occurrences = List(('q',2), ('s',1), ('z',1))
occurrences: List[(Char, Int)] = List((q,2), (s,1), (z,1))

scala> val occurrences2 = List(('q',1), ('z',1))
occurrences2: List[(Char, Int)] = List((q,1), (z,1))

scala> def combinations(occurrences: Occurrences):
List[Occurrences] = occurrences match {
    |     case Nil => List(List())
    |     case ((w,n)::o) =>
    |       val comb=combinations(o)
    |       comb ::: {for {
    |       c <-comb
    |       i <- 1 to n}
    |       yield (w,i)::c}
    | }
combinations: (occurrences: Occurrences)List[Occurrences]

scala> combinations(occurrences)
res32: List[Occurrences] = List(List(), List((z,1)), List((s,1)),
List((s,1), (z,1)), List((q,1)), List((q,2)), List((q,1), (z,1)), List((q,2),
(z,1)), List((q,1), (s,1)), List((q,2), (s,1)), List((q,1), (s,1), (z,1)),
List((q,2), (s,1), (z,1)))

scala> occurrences.toSet.subsets.foreach(println)
Set()
Set((q,2))
Set((s,1))
Set((z,1))
Set((q,2), (s,1))
Set((q,2), (z,1))
Set((s,1), (z,1))
Set((q,2), (s,1), (z,1))

scala> def subtract(x: Occurrences, y: Occurrences):
Occurrences = y match {
    |     case Nil => x
    |     case (w,n)::o => subtract(for {e <- x if (e._1!=w || e._2!=n)}
yield (e._1,if (e._1==w) e._2-n else e._2),o)
    | }
subtract: (x: Occurrences, y: Occurrences)Occurrences

scala> subtract(occurrences,occurrences2)
res34: Occurrences = List((q,1), (s,1))

scala> val occurrences =List(('e',2), ('i',1), ('j',1), ('m',1), ('n',1),
('o',1), ('t',1), ('u',1), ('z',1))
occurrences: List[(Char, Int)] = List((e,2), (i,1), (j,1), (m,1), (n,1),
(o,1), (t,1), (u,1), (z,1))

scala> def sentenceAnagrams_aux(occ: Occurrences):
List[Sentence]= occ match {
    |     case Nil => List(List())
    |     case _ =>{
    |       val c=combinations(occ)
    |       for {(o,words) <- dictionaryByOccurrences
    |         if (c contains o)
    |         s <- sentenceAnagrams_aux(subtract(occ,o))
    |         w <- words}
    |         yield w::s
    |     }.toList
    | }
sentenceAnagrams_aux: (occ: Occurrences)List[Sentence]

scala> def sentenceAnagrams(sentence: Sentence):
List[Sentence] =
sentenceAnagrams_aux(sentenceOccurrences(sentence))
sentenceAnagrams: (sentence: Sentence)List[Sentence]

scala> sentenceAnagrams(List("burtoncourt"))
res35: List[Sentence] = List(List(burtoncourt), List(cuon, rott, bru),
List(cuon, bru, rott), List(corn, butot, ur), List(corn, ur, butot),
List(trun, orto, buc), List(trun, troo, buc), List(trun, orto, buc),
List(trun, troo, buc), List(trun, buc, orto), List(trun, buc, orto),
List(trun, buc, troo), List(trun, buc, troo), List(rott, cuon, bru),
List(rott, bru, cuon), List(orto, trun, buc), List(troo, trun, buc),
List(orto, trun, buc), List(troo, trun, buc), List(orto, buc, trun),
List(troo, buc, trun), List(orto, buc, trun), List(troo, buc, trun),
List(buc, trun, orto), List(buc, trun, orto), List(buc, trun, troo),
List(buc, trun, troo), List(buc, orto, trun), List(buc, orto, trun),
List(buc, troo, trun), List(buc, troo, trun), List(octon, bru, urt),
List(octon, urt, bru), List(bru, cuo...
```