Un codage de Huffman est un algorithme de compression des chaînes de caractères. Dans un texte non-compressé, chaque caractère est représenté par le même nombre de bits, souvent huit. Un codage de Huffman assigne à chaque caractère une représentation de longueur différente, suivant s'il est commun ou non. Bien entendu, un codage donné n'est optimal que pour un seul texte.

Conctrètement un codage de Huffman peut être représenté par un arbre binaire dont les feuilles sont les caractères à encoder. Chaque nœud est un ensemble contenant les caractères présents dans les feuilles en dessous. De plus, chaque caractère dispose d'un poids — sa fréquence dans le texte — et chaque nœud est annoté du poids total des feuilles en dessous, une information nécessaire lors de la construction de l'arbre. La figure 1 est un arbre de codage pour un texte où "A" a la fréquence 8, "B" la fréquence 3 et tous les autres caractères la fréquence 1.

Pour un arbre de codage donné, on obtient l'encodage d'un caractère en traversant l'arbre de la racine à la feuille contenant le caractère. En chemin, chaque fois que l'on prend une branche à gauche, on ajoute 0 à la représentation, chaque fois que l'on prend une branche à droite, on ajoute 1. Ainsi, l'arbre de la figure 1 encode le caractère "D" comme "1011".

Le décodage commence à la racine de l'arbre. On lit successivement les bits de la séquence à décoder ; pour chaque 0 on descend dans l'arbre à gauche, pour chaque 1 à droite. Lorsqu'on atteint une feuille, on a décodé le caractère correspondant et l'on recommence à la racine. Ainsi, la séquence de bits "10001010" se décode avec l'arbre de la figure 1 en "BAC".

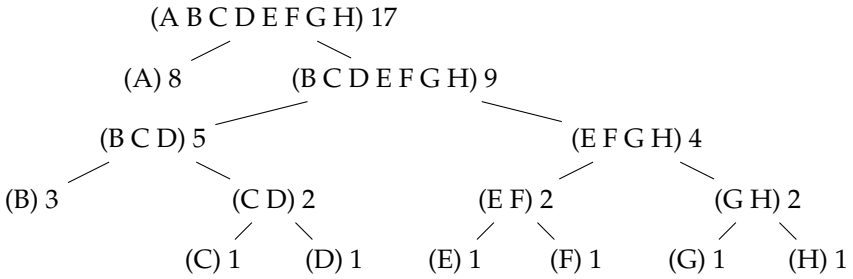

FIGURE 1 – Un codage de Huffman encodé comme un arbre

Mise en place

En Scala, un arbre de codage de Huffman est représenté par le type algébrique suivant.

```
abstract class CodeTree
case class Fork (
  left: CodeTree,
  right: CodeTree,
```

```
  chars: List[Char],
  weight: Int
) extends CodeTree
case class Leaf(char: Char, weight: Int) extends CodeTree
```

Pour commencer, définissez les deux fonctions suivantes :

1. `weight` qui retourne la poids total d'un arbre de codage donné.
   ```
   def weight(tree: CodeTree): Int = tree match ...
   ```
2. `chars` qui retourne la liste des caractères définis dans un arbre de codage donné.
   ```
   def chars(tree: CodeTree): List[Char] = tree match ...
   ```

A l'aide de ces fonctions, il est possible de définir `makeCodeTree`, une fonction qui facilite la création d'un arbre de codage en calculant automatiquement la liste des caractères et le poids lors de la création d'un nœud.

```
def makeCodeTree(left: CodeTree, right: CodeTree) =
  Fork (
    left, right, chars(left) ::: chars(right),
    weight(left) + weight(right)
  )
```

`makeCodeTree` s'utilise de la façon suivante :

```
val sampleTree = makeCodeTree(
  makeCodeTree(Leaf('x', 1), Leaf('e', 1)),
  Leaf('t', 2)
)
```

Création d'arbres de codage

Etant donné un texte, il est possible de calculer un arbre de codage optimal dans le sens où l'encodage de ce texte sera de longueur minimum — en gardant toute l'information.

Afin d'obtenir un tel arbre optimal à partir d'une liste de caractères, il vous faut maintenant définir une fonction `createCodeTree` ayant la signature suivante :

```
def createCodeTree(chars: List[Char]): CodeTree = ...
```

Pour ce faire, nous vous suggérons de procéder ainsi :

1. Commencez par écrire une fonction `times` qui calcule la fréquence de chaque caractère dans le texte :
   ```
   def times(chars: List[Char]): List[(Char, Int)] = ...
   ```
2. Ecrivez ensuite une fonction `makeLeafList` qui génère une liste contenant toutes les feuilles de l'arbre de codage à construire — le cas `Leaf` du type algébrique `CodeTree` — ordonnées par poids croissant, le poids d'une feuille étant la fréquence d'apparition de son caractère.
   ```
   def makeLeafList(freqs: List[(Char, Int)]):
     List[CodeTree] = ...
   ```

3. Ecrivez une fonction `singleton` qui vérifie si une liste contient un seul et unique arbre.
   ```
   def singleton(trees: List[CodeTree]): Boolean = ...
   ```
4. Ecrivez une fonction `combine` qui enlève les deux arbres de poids le plus faible de la liste construite à l'étape précédente et les fusionne en un seul arbre au moyen d'un nœud `Fork`. Ajoutez ce nouvel arbre à la liste — maintenant plus courte d'un élément — en préservant l'ordre.
   ```
   def combine(trees: List[CodeTree]): List[CodeTree] =
     ...
   ```
5. Ecrivez une fonction `until` qui fait appel aux deux fonctions définies plus haut jusqu'à ce que la liste ne contienne plus qu'un seul arbre. Cet arbre est l'arbre de codage optimal. La fonction `until` s'utilise de la manière suivante :
   ```
   until(singleton, combine)(trees)
   ```
   où l'argument `trees` a le type `List[CodeTree]`.
6. Finalement utilisez les fonctions définies ci-dessus pour implanter la fonction `createCodeTree` en respectant la signature indiquée plus haut.

Décodage

Définissez la fonction `decode` qui décode une liste de bits encodés par un codage de Huffman, étant donné l'arbre de codage correspondant.

```
type Bit = Int
def decode(tree: CodeTree, bits: List[Bit]): List[Char] =
  ...
```

Encodage ...

Cette section concerne l'encodage de Huffman d'une séquence de caractères en un séquence de bits.

...par arbre de Huffman

Définissez la fonction `encode` qui encode une liste de caractères à l'aide d'un encodage de Huffman, étant donné un arbre de codage.

```
def encode(tree: CodeTree)(chars: List[Char]): List[Bit] =
  ...
```

Votre implantation traversera l'arbre de codage pour chaque caractère, une tâche que vous aurez tout intérêt à abstraire dans une fonction annexe.

...par table de codage

La fonction précédente est simple, mais peu efficace. Votre but est maintenant de définir `quickEncode`, une fonction équivalente à `encode`, mais plus efficace.

```
def quickEncode(tree: CodeTree)(chars: List[Char]):
  List[Bit] = ...
```

Votre implantation construira une seule fois une table d'encodage qui, pour chaque caractère possible, donne la liste de bits l'encodant. La façon la plus simple — mais pas la plus efficace — est d'encoder la table de caractères comme une liste de paires.

```
type CodeTable = List[(Char, List[Bit])]
```

L'encodage se fait ensuite en accédant à cette table, par exemple à l'aide de la fonction `codeBits`.

```
def codeBits(table: CodeTable)(char: Char): List[Bit] =
  ...
```

La création de la table est définie par `convert` qui traverse l'arbre de codage en construisant la table des caractères au fur et à mesure.

```
def convert(t: CodeTree): CodeTable = ...
```

Implantez la fonction `convert` au moyen de la fonction `mergeCodeTables` ci-dessous :

```
def mergeCodeTables(a: CodeTable, b: CodeTable):
  CodeTable = ...
```

Bonus : Utilisez la méthode `map` de la classe `List` pour implanter la fonction `mergeCodeTables`.