Spectrum Analysis

NAME: COURSE:

In this exercise we build a simple spectrum analyser. For this purpose, we first implement the discrete Fourier transform.

Discrete Fourier Transform Given a sequence $x_0, \ldots, x_{N-1}$ of real coefficients (representing a signal), its Fourier transform is a sequence of *complex* coefficients $X_0, \ldots, X_{N-1}$, defined as:

$$X_n = \sum_{k=0}^{N-1} x_k e^{-2\pi \frac{kn}{N} i} \in \mathbb{C} \tag{1}$$

for $n = 0, 1, \ldots, N-1$.

On the other hand, given a sequence of Fourier coefficients $X_0, \ldots, X_{N-1}$, the inverse Fourier transform of this sequence is a sequence of real coefficients $x_0, \ldots, x_{N-1}$ where:

$$x_k = \frac{1}{N} \sum_{n=0}^{N-1} X_n e^{2\pi \frac{kn}{N} i} \in \mathbb{R} \tag{2}$$

Note that these definitions use complex numbers. In Haskell, we use the data type `Complex` from the module `Data.Complex` to represent complex numbers. This data type is defined as:

```
data Complex a = a :+ a
```

That is, it represents a complex number by its real and imaginary parts. Thus, we write a a complex number $a + bi$ as `a :+ b`. The usual operations, such as, `+`,`-`,`*`,`/`, `exp`, etc. are also defined for this data type.

To make the assignment more convenient, we first define two auxiliary functions to convert a `Double` into a `Complex Double`.

Exercise 1 Write two functions, `d2rc :: Double -> Complex Double` en `d2ic :: Double -> Complex Double` that accept a `Double` and put it in the real or the imaginary part of a `Complex Double`, respectively.

```
> d2rc 2.0
2.0 :+ 0.0

> d2ic 2.0
0.0 :+ 2.0

> d2ic 2.0 + d2rc 2.0
2.0 :+ 2.0
```

Exercise 2 We also use a data type `Fourier` to distinguish Fourier coefficients from real coefficients. The type synonym `type Signal = [Double]` has already been predefined.

Now complete the definition of the data type `Fourier` such that it contains lists of coefficients of type `Complex Double`. Also complete the definitions of `mkFourier` and `unFourier`, to wrap and unwrap a list into a `Fourier` data type.

```
> unFourier $ mkFourier [1 :+ 2, 3 :+ 4, d2ic 4]
[1.0 :+ 2.0,3.0 :+ 4.0,0.0 :+ 4.0]
```

Aside: Comparing floating-point numbers

The result of a floating-point computation is almost never exact. Therefore, when comparing two numbers, we never compare exactly. Instead, we look at the difference in absolute value.

Exercise 3 The predefined class `AlmostEq` has a method `(~=) :: a -> a -> Bool` that is `True` if two values are almost equal.

Now define:

1. An instance of `AlmostEq` for `Double` such that:

$$a \sim= c \iff |a - c| < \varepsilon$$

   where $\varepsilon = 10^{-14}$.

   Hint The function `abs` can be useful. In Haskell, 10^{-14} is written as `1e-14`.

2. An instance of `AlmostEq` for `Complex Double` such that:

$$a + bi \sim= c + di \iff |(a + bi) - (c + di)| < \varepsilon$$

   where $\varepsilon = 10^{-14}$. Note that $|(a + bi) - (c + di)|$ is a real number. [1]

   Hint The function `magnitude` can be useful.

3. An instance of `AlmostEq a => AlmostEq [a]` that is true if all coordinates are pairwise almost equal and the lists have identical lengths.

4. An instance of `Fourier` that is true if all coordinates are pairwise equal.

   Hint use the instance of `[a]`.

```
> 1.0 :+ (0 :: Double) ~= 1.0 :+ 0
True
> 1e-16 :+ (0 :: Double) ~= 0 :+ 0
True
> 1e-14 ~= (0.0 :: Double)
False
> [1.0, (0 :: Double)] ~= [1.0, 1e-16]
True
> [1.0, (0 :: Double)] ~= [1.0]
False
> mkFourier [1.0 :+ 0, 1e-16 :+ 0] ~= mkFourier [1.0 :+ 0, 0 :+ 0]
True
> mkFourier [1.0 :+ 0, 1e-16 :+ 0] ~= mkFourier [1.0 :+ 0, 1e-13 :+ 0]
False
```

Naive DFT

In this section you need to implement the DFT and its inverse. **Be careful!** Forgetting to factors such as (-1) of $\frac{1}{N}$ may lead to errors that may be hard to debug.

[1] This is called the *magnitude* or the absolute value of the complex number.

Exercise 4 Implement the DFT directly using Equation 1. Complete the definition of `dft :: Signal -> Fourier` such that

$$\texttt{unFourier (dft [}x_0\texttt{,...,}x_{N-1}\texttt{]) = [}X_0\texttt{,...}X_{N-1}\texttt{]}$$

Hint Try to use list comprehensions. The functions `exp`, `pi` and `sum` are useful here.

```
> unFourier (dft [1,2,2,1]) ~= [6 :+ 0, (-1) :+ (-1), 0:+ 0, (-1) :+ 1]
True
> unFourier (dft [])
[]
```

Exercise 5 Implement the inverse DFT directly using Equation 2. Complete the defintion of `idft :: Fourier -> Signal` such that

$$\texttt{idft (mkFourier [}X_0\texttt{,...,}X_{N-1}\texttt{]) = [}x_0\texttt{,...}x_{N-1}\texttt{]}$$

Hint Try to use list comprehensions. The functions `exp`, `pi`, `sum` and `realPart` are useful here.

```
> idft (mkFourier [6 :+ 0, (-1) :+ (-1), 0:+ 0, (-1) :+ 1]) ~= [1,2,2,1]
True
> idft (dft [sin t | t <- [0,pi/4..2*pi]]) ~= [sin t | t <- [0,pi/4..2*pi]]
True
```

Spectrum Analyser

A spectrum analyser is a device that shows the energy levels of the frequency spectrum of a signal. Concretely, the energy of the frequency $\frac{n}{N}$ is simply the magnitude of the Fourier coefficient X_n.

Exercise 6 Write a function `spectrumAnalyser :: Signal -> IO ()` which shows the frequency spectrum of a signal by printing a line of '#' symbols for every coefficient. The number of '#' symbols corresponds to the magnitude of the coefficient.

Proceed as follows: first, compute the Fourier transform of the input signal, then compute the magnitudes of all coefficients and rescale them, making the largest magnitude equal to 40. This value, rounded up to the nearest integer, is the number of of '#' symbols to print.

Hint The functions `magnitude` (from `Data.Complex`), `ceiling` and `maximum` are useful here.

```
> spectrumAnalyser [1,2,2,1]
########################################
##########

##########
> spectrumAnalyser [1,2,1,2]
########################################

##############

> spectrumAnalyser [sin t + sin (2*t) | t <- [0,pi/4..2*pi-pi/4]]
#
########################################
########################################
#
#
```

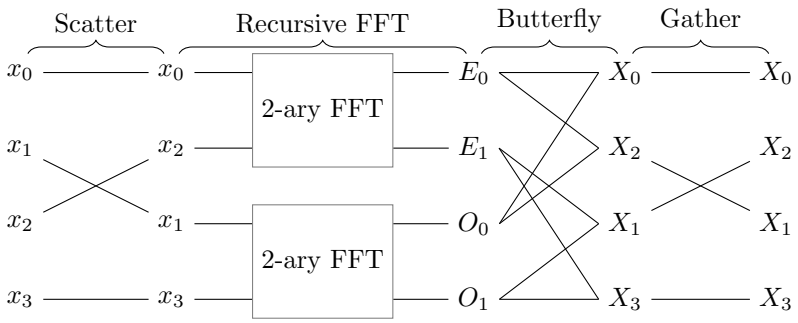

Figure 1: The Cooley-Tukey algorithm, for $N = 4$.

```
#
########################################
########################################
```

Fast Fourier Transform (Cooley-Tukey)

In practice, the performance of the naive DFT (directly implementing Equations 1 and 2) leaves something to be desired. In fact, its asymptotic time complexity is $\mathcal{O}(N^2)$. For this reason, practical implementations usually resort to another class of algorithms, the so-called *Fast Fourier Transforms* (FFT), of which the Cooley-Tukey algorithm is probably the most well known. Here we use this algorithm for the specific case where N is *a power of two*.

We solve the FFT recursively, by dividing the coefficients in the evenly and oddly numbered ones and computing the Fourier transform of each set separately. Given these coefficients, we can derive the coefficients of the full transform in linear time. This improves the time complexity to $\mathcal{O}(N log\ N)$.

The algorithm consists of a number of phases (illustrated in Figure 1):

1. *Scatter*: Divide the coefficients in even and odd coefficients $x_0, x_2, \ldots, x_{N-2}$ and $x_1, x_3, \ldots, x_{N-1}$.

2. *Recursive FFT*: Compute the Fourier transforms of the even and odd sublists:

$$[E_0, \ldots, E_{\frac{N}{2}-1}] = \texttt{unFourier (fft [}x_0, x_2, \ldots, x_{N-2}\texttt{])}$$
$$[O_0, \ldots, O_{\frac{N}{2}-1}] = \texttt{unFourier (fft [}x_1, x_2, \ldots, x_{N-1}\texttt{])}$$

3. *Butterfly*: Combine the even and odd transforms to obtain the full transform according to the formula:

$$X_n = E_n + e^{-2\pi \frac{n}{N} i} O_n$$
$$X_{n+\frac{N}{2}} = E_n - e^{-2\pi \frac{n}{N} i} O_n$$

   for $n = 0, \ldots, \frac{N}{2} - 1$. The factor $e^{-2\pi \frac{n}{N} i}$ is often called the *twiddle* factor, which is shared between the computations for X_n and $X_{n+\frac{N}{2}}$.

4. *Gather*: Put the coefficients back into the right order: the output of the Butterfly phase puts X_n at index $2n$, and $X_{\frac{N}{2}+n}$ at index $2n + 1$ for $0 \leq n < \frac{N}{2}$. Shuffle the coefficients such that coefficient X_n is at index n for $0 \leq n < N$.

Exercise 7 Implement the function `scatter :: [a] -> ([a],[a])` such that the evenly indexed elements occur in the first list, and the oddly numbered elements in the second list. Assume that the list contains an even number of elements.

```
> scatter [0,1,2,3,4,5,6,7]
([0,2,4,6],[1,3,5,7])
> scatter [1,2,3,4,5,6,7,8]
([1,3,5,7],[2,4,6,8])
> scatter [False,True,False,True]
([False,False],[True,True])
> scatter [] :: ([Double],[Double])
([],[])
```

Also implement the function `gather :: [a] -> [a]` which returns the elements to their correct positions. Note `gather` can be implemented by performing `scatter` and appending the resulting lists. This is a consequence of those X_n belonging in the second half of the final list (i.e. those where $n \geq \frac{N}{2}$) being at the oddly numbered positions. Figure 1 also shows this visually: the lines in the Scatter and Gather phases are identical. Assume that the list contains an even number of elements.

```
> gather [0,4,1,5,2,6,3,7]
[0,1,2,3,4,5,6,7]
```

Exercise 8 Implement `twiddle :: Int -> Int -> Complex Double -> Complex Double -> [Complex Double]` such that

$$\texttt{twiddle } N \ n \ E_n \ O_n \texttt{ = [}E_n + e^{-2\pi \frac{n}{N} i} O_n\texttt{, } E_n - e^{-2\pi \frac{n}{N} i} O_n\texttt{]}$$

```
> twiddle 16 0 (1 :+ 0) ((-1) :+ 1)
[0.0 :+ 1.0,2.0 :+ (-1.0)]
> -- note: exp (0 :+ (-2*pi*1/4)) = -i
> twiddle 4 1 1 ((-1) :+ 1)
[2.0 :+ 1.0,1.1102230246251565e-16 :+ (-1.0)]
```

Exercise 9 Implement `butterfly :: Int -> Fourier -> Fourier -> [Complex Double]` which executes the butterfly step, given N and the Fourier transforms of the even and odd elements. For example, for $N = 8$ (for the sake of readability, we elide `mkFourier` in the arguments to `butterfly`):

$$\texttt{butterfly 8 [}E_0\texttt{,}E_1\texttt{,}E_2\texttt{,}E_3\texttt{] [}O_0\texttt{,}O_1\texttt{,}O_2\texttt{,}O_3\texttt{] = [}X_0\texttt{,}X_4\texttt{,}X_1\texttt{,}X_5\texttt{,}X_2\texttt{,}X_6\texttt{,}X_3\texttt{,}X_7\texttt{]}$$

Hint Apply `twiddle` elementwise, and concatenate the results. The function `zipWith3` can be useful for this purpose.

```
> butterfly 4 (mkFourier [3 :+ 0, (-1) :+ 0]) (mkFourier [3 :+ 0, 1 :+ 0])
[6.0 :+ 0.0,0.0 :+ 0.0,(-0.9999999999999999) :+ (-1.0),(-1.0) :+ 1.0]
```

Exercise 10 Implement the function `fft :: Signal -> Fourier` that combines the different phases of the FFT: First use `scatter` to split the input signal, then recursively call the FFT, use `butterfly` to recombine the signals, and put the coefficients back in their correct positions with `gather`.

Assume that the number of input coefficients is a strictly positive power of two. Note that by recursively applying the FFT you eventually obtain a signal that has only a single coefficient x_k, by the definition of the DFT, you can see that $X_k = x_k + 0i$ (see also the example below).

```
> unFourier (fft [1])
[1.0 :+ 0.0]
> unFourier (fft [1,2,2,1]) ~= unFourier (dft [1,2,2,1])
True
> idft (fft [1,2,2,1]) ~= [1,2,2,1]
True
> dft [sin t | t <- [0,pi/4..2*pi-pi/4]] ~= fft [sin t | t <- [0,pi/4..2*pi-pi/4]]
True
```

Optional Adapt `fft` such that the algorithm works for any input size: use the Cooley-Tukey algorithm if N is even, otherwise fall back to the naive DFT.

Good luck!