The purpose of this Literate Haskell program is to implement a function that does global sequence alignment using Needleman/Wunsch techniques.\footnote{1}

The algorithm is based on two steps: first, filling a matrix with the maximal alignment scores for each element and then tracing a path connecting the top-left and the bottom-right cell. Note that the matrix is $O(n \cdot m)$ memory-wise and therefore pretty inefficient, you don’t want to use this on bigger sequences.

In Haskell, a good way to implement Dynamic Programming like this is an array that will memoize a lazy stream of scores per cell. This allows $O(1)$-lookup of formerly calculated values without losing referential transparency and (to an extent) lazy evaluation.

```haskell
import Array

align :: String -> String -> [String]
align da db = format $ reverse $ traceback lena lenb
  where
    lena = length da
    lenb = length db
```

The algorithm is easier to express when the sequences to align are one-indexed, since the borders of the matrix are used as special values. An easy way to achieve this is prepending a space:

```haskell
a = '␣' : da
b = '␣' : db
```

\footnote{The author can be reached at \url{http://chneukirchen.org}}

\footnote{More about these techniques, graphics helpful for understanding, and a codeless step-by-step explanation can be found at \url{http://www.sbc.su.se/~pjk/molbioinfo2001/dynprog/dynamic.html}}
memscore is the array that contains the actual matrix. It is filled using a lazy stream of scores for each element.

\[
\text{memscore} = \text{listArray} \left( (0,0), (\text{lena}, \text{lenb}) \right)
\]

\[
[\text{score } x \ y \mid x \leftarrow [0..\text{lena}], y \leftarrow [0..\text{lenb}]]
\]

The scoring function looks very confusing since Haskell's array access operator is not very elegant. I'll introduce an infix operator \(i \mathbin{\@\@} j\) that corresponds to \(M_{i,j}\):

\[
\text{infix } 5 \mathbin{\@\@}
\]

\[
(\mathbin{\@\@}) \ i \ j = \text{memscore}! (i, j)
\]

The score \(M_{i,j}\) of each element is determined in below code as follows, the borders of the matrix with \(i = 0\) and \(j = 0\) are initialized to zero. (More complex scoring algorithms could be added easily.)

\[
M_{i,j} = \text{maximum of } \begin{cases} M_{i-1,j-1} + S_{i,j} \\ M_{i,j-1} + w \\ M_{i-1,j} + w \end{cases}
\]

The gap penalty \(w\) is zero here for reasons of simplicity.

\[
\text{score } 0 \_ = 0 \\
\text{score } \_ 0 = 0 \\
\text{score } x \ y = \text{maximum} \left[ (x - 1 \mathbin{\@\@} y - 1) + \text{difference } x \ y, \\ x - 1 \mathbin{\@\@} y, \\ x \mathbin{\@\@} y - 1 \right]
\]

\(S_{i,j}\) is a mismatch penalty defined here like this:

\[
S_{i,j} = \begin{cases} 0 & \text{if the symbols at position } i \text{ and position } j \text{ match} \\ 1 & \text{otherwise} \end{cases}
\]

where \(\text{difference } x \ y \mid a \equiv b \equiv y = 1 \\ \mid \text{otherwise } = 0\)

\(\text{traceback}\) now finds the path connecting both corners of the matrix and collects the appropriate symbols (or spaces for gaps).

\[
\text{traceback} :: \text{Int} \to \text{Int} \to [(\text{Char}, \text{Char})]
\]

\[
\text{traceback} 0 \ 0 = [] \\
\text{traceback} \ x \ y \mid x \equiv 0 = (', u' , b \equiv y) : \text{traceback} \ 0 \ (y - 1) \\
\mid y \equiv 0 = (a \equiv x, ', u' ) : \text{traceback} \ (x - 1) \ 0 \\
\mid x \mathbin{\@\@} y \equiv x \mathbin{\@\@} y - 1 = (', u' , b \equiv y) : \text{traceback} \ x \ (y - 1) \\
\mid x \mathbin{\@\@} y \equiv x - 1 \mathbin{\@\@} y = (a \equiv x, ', u' ) : \text{traceback} \ (x - 1) \ y \\
\mid \text{otherwise} = (a \equiv x, b \equiv y) : \text{traceback} \ (x - 1) \ (y - 1)
\]
The resulting list of tuples like [(‘a’, ‘d’), (‘b’, ‘e’), (‘c’, ‘f’)] gets converted by format into ["abc", "def"].

\[
\text{format } l = [\text{map fst } l, \text{map snd } l]
\]

Finally, a small main program to test the algorithm:

\[
dna1 = "GAATTCAGTTA"
dna2 = "GGATCGA"
main = mapM putStrLn $ align dna1 dna2
\]

Expected output:

G AATTCAGTTA
GGA T C G A

As you can see, corresponding symbols are aligned, with appropriate gaps in between. Implementing more complex rules for alignment is left as an exercise for the reader.

A run where bigger gaps are needed:

\[
dna1 = "ATGGCTTCTACC"
dna2 = "TATCAAAAGCCG"
\]

ATGGCTTCTA CC
TAT C AAAAGCCG

As you can see, corresponding symbols are aligned, with appropriate gaps in between. Implementing more complex rules for alignment is left as an exercise for the reader.