Introduction to Programming: Assignment 3

Due: October 18, 2022. 11.55 pm

Important Instructions:

Submit your solutions in three files named `emailid.1.hs`, `emailid.2.hs` and `emailid.3.hs`, on Moodle. `emailid` is your CMI email id, without the `@cmi.ac.in`. For example, if I were to submit the solutions, I would submit it as `spsuresh.1.hs`, `spsuresh.2.hs` and `spsuresh.3.hs`. I am attaching template files for each, with the name of the function and its type, and a rudimentary definition. You have to replace `undefined` with the correct solution. You can use any number of auxiliary definitions in the same file. Finally rename the files (by changing `template` to your email id) and submit. I am also attaching a separate file for each problem with the test cases. You can add your own test cases and check.

---

1. Knights are chess pieces whose moves are characterized as two and a half squares – they move two squares in one direction and one square in an orthogonal direction. On a chessboard, rows are called *ranks* and columns are called *files*. The square on the x^{th} rank and the y^{th} file is represented by the pair (x, y). The set of squares on an 8×8 chessboard is thus given by

   ```
   squares = [(x,y) | x ← [0..7], y ← [0..7]]
   ```

   Define a function

   ```
   knightMove :: (Int, Int) → Int → [(Int, Int)]
   ```

   with the behavior that

   ```
   knightMove (x, y) n
   ```

   gives the list of squares where the knight could be in after a total of `n` moves, starting from the square `(x,y)`. Make sure that your list has no duplicates, and is lexicographically sorted.

   Sample cases:

   ```
   knightMove (0,0) 0      = [(0,0)]
   knightMove (0,0) 1      = [(1,2),(2,1)]
   knightMove (0,0) 3      = [(0,1),(0,3),(0,5)
   ```

```
                          ,(1,0),(1,2),(1,4),(1,6)
                          ,(2,1),(2,3),(2,5)
                          ,(3,0),(3,2),(3,4),(3,6)
                          ,(4,1),(4,3),(4,5)
                          ,(5,0),(5,2),(5,4)
                          ,(6,1),(6,3)
                          ]
knightMove (2,2) 2      = [(0,2),(0,6)
                          ,(1,1),(1,3),(1,5)
                          ,(2,0),(2,2),(2,4),(2,6)
                          ,(3,1),(3,3),(3,5)
                          ,(4,2),(4,6)
                          ,(5,1),(5,3),(5,5)
                          ,(6,0),(6,2),(6,4)
                          ]
knightMove (2,2) 1000   = [(0,0),(0,2),(0,4),(0,6)
                          ,(1,1),(1,3),(1,5),(1,7)
                          ,(2,0),(2,2),(2,4),(2,6)
                          ,(3,1),(3,3),(3,5),(3,7)
                          ,(4,0),(4,2),(4,4),(4,6)
                          ,(5,1),(5,3),(5,5),(5,7)
                          ,(6,0),(6,2),(6,4),(6,6)
                          ,(7,1),(7,3),(7,5),(7,7)
                          ]
```

2. This problem concerns editing a string to arrive at another string. We start at the left-most letter, and either keep it and skip to the next letter, or modify the letter, or insert a new letter at that position, or delete the letter, and continue till the string is completely transformed. The series of edits involved in the transformation is summarised by an *edit string* consisting of the characters `-`, `i`, `d`, and `m`. For instance, here is an edit string that transforms `kitten` to `sitting`.

```
 kitten
s itting
id---m-i
```

   Each edit operation has a cost. Skipping a letter has cost 0, modifying a letter has cost 1, while insert and delete have cost 2. One can see that the above sequence of edits has total cost 7. Given below is a sequence of edits with total cost 4.

```
kitten
sitting
m---m-i
```

One can check that the sequence of edits represented by the edit string `"iiiiiiidddddd"` has cost 26.

The task is to find an edit string with minimum overall cost to go from one string to another. (This cost is called the *edit distance*.) Define a function

```
editDistance :: String → String → (Int, String)
```

such that `editDistance as bs` returns `(n,s)`, where `n` is the edit distance, and `s` is an edit string with cost `n`. For instance, the edit distance between `"kitten"` and `"sitting"` is 4.

A naïve recursive program to compute edit distance would take exponential time in the worst case, because of repeated recursive calls with same arguments. You should program using arrays to perform the task in polynomial time.

Sample cases:

```
editDistance "kitten" "sitting"
    = (4,"m---m-i")
editDistance "Kareem is the scoring leader"
    "LeBron will be the scoring leader"
    = (17,"mmmmmm-i-mi-iii------------------")
editDistance "Jordan was an all-time great"
    "LeBron is even greater"
    = (28,"mmmmm--md--mmmmmmmmmm-dd-ddd")
editDistance "LeBron is even greater"
    "Jordan was an all-time great"
    = (28,"mmmmm--mi--mmmmmmmmmm-ii-iii")
editDistance "" "This is a nonempty string"
    = (50,"iiiiiiiiiiiiiiiiiiiiiiii")
editDistance "This is a nonempty string" ""
    = (50,"ddddddddddddddddddddddddd")
editDistance "bzrgfczhabmvneg" "zgbbbfeqnxbddfudfjjp"
    = (23,"mmmmi-mmmm-mmmmmiiii")
editDistance "fittingly" "misfitting"
    = (9,"m-mm-mmmmi")
editDistance "fittingly" "unfitting"
    = (8,"ii------dd")
```

3. The next program involves matrix chain multiplication. We are given a sequence (chain) of matrices $(M_0, M_1, \ldots, M_n)$, where $n \geqslant 0$. We wish to compute their product

$$M_0 M_1 \cdots M_n.$$

When $n = 0$, the product is just M_0. Let us assume that the dimensions of each matrix M_i is $r_i \times c_i$. Note that the multiplication can be carried out if $r_{i+1} = c_i$ for each $i \in [0 \cdots (n-1)]$. We are interested not in the final result itself, but in the cost of the chain multiplication. To compute MM' where M is an $(r \times c)$-matrix, and N is an $(r' \times c')$-matrix (with $c = r'$), it takes rcc' scalar multiplications.

Since matrix multiplication is associative, we can carry out the multiplication in any order. But some orders of multiplication are more efficient than others. For example, if M_0, M_1 and M_2 are of dimensions 10×100, 100×5 and 5×50 respectively, multiplying in the order $(M_0 M_1)M_2$ requires a total of $5000 + 2500 = 7500$ scalar multiplications, while multiplying in the order $M_0(M_1 M_2)$ requires a total of $25000 + 50000 = 75000$ scalar multiplications. Clearly the first order of multiplication is more efficient.

Given a chain of matrices of compatible dimensions, we wish to compute the optimal number of scalar multiplications needed to compute the chain product, and also the order in which to multiply them. For simplicity we present the input as a list of $n + 1$ integers $[r_0, \ldots, r_{n+1}]$ (where $n \geqslant 0$), with $r_i \times r_{i+1}$ being the dimensions of the matrix M_i.

Define a function

```
chainOrder :: [Int] → (Int, String)
```

which computes the optimal cost and optimal chain order.

Sample cases (remember that the input list will have at least 2 entries):

```
chainOrder [500,1000]       = (0,"")
                    -- only one matrix, nothing to multiply
chainOrder [1,4,5,3,1]      = (38,"(((M0 * M1) * M2) * M3)")
chainOrder [1,4,5,3,4,1]    = (50,"(((M0 * M1) * M2) * (M3 * M4))")
chainOrder [96,63,25,96,72] = (496800,"((M0 * M1) * (M2 * M3))")
chainOrder [46,49,10,3,69]  = (17754,"((M0 * (M1 * M2)) * M3)")
```

We want this program to work for even larger sample cases than the above, so remember to use arrays in your solution.