Question 1:

```
function ComputeExpression(str)
   init(DigitStack), init(OprtStack)
   ValidDigit = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')
   ValidOperator = ('(', ')', '+', '-', '/', '*')
   LastCharDigit<-False
   for i<-0 to length(str)-1 do
      if (str[i] in ValidDigit) and !LastCharDigit then
         push(DigitStack, str[i])
         LastCharDigit<-True
      elseif str[i] in ValidOperator then
         push(OprtStack, str[i])
         LastCharDigit<-False
      else then
         return NotWellFormed
      if str[i] == ')' then
         pop(OprtStack)
         if len(DigitStack) < 2 or len(OprtStack) < 2 then
            return NotWellFormed
         A<-pop(DigitStack), O<-pop(OprtStack),B<-pop(DigitStack)
         if O not equal to '(' or ')' then
            total <- apply A with the opperator O by B
            push(DigitStack, total)
         else then
            return NotWellFormed
         if pop(OprtStack) not equal to '(' then
            return NotWellFormed
   if len(DigitStack) == 1 and len(OprtStack) == 0 then
      return pop(DigitStack)
   else then
      return NotWellFormed
```

Question 3b

The algorithm used in 3a is a variation of a topological sort. First the data is organised into tree data structures, which have integer values for tree number/index and amount of incoming edges, and a deque for the indices of the trees children.

The program takes in the first line which gives the total amount of edges(m) as well as vertices(n), this information is used to create an array of tree pointers with length n. The array is then populated with empty trees and the program begins to read the edges. The parent of each edge has the child's index/tree number pushed onto the deque of outgoing edges, while the child's incoming counter is incremented by 1. This leaves total space occupied by the program at O(n+m) complexity given n trees with a total of m items in deques as well as the time taken being m.

After the program has all of the edges processed, the max_path_len() function is called on the array of trees to determine if there is an unbroken path down the hill. Here is the pseudo code for the function:

```
Bool max_path_len(array)
   Current_vertex = array[0]
   For i=1 to len(array)// including
      If current_vertex.outgoing is empty
         break
      found<-false
      for child in current_vertex.outgoing
         child.incoming -= 1
         if !child.incoming
            found<-true
            current_vertex<- child
      if !found
         return false
   if I == len(array)
      return true
   return false
```

The algorithm starts at Tree 0 and identifies the next ordered tree by the amount of incoming edges. This is because if a node has only one incoming edge in a directed acyclic graph it either is the next node in the complete path or the path is impossible to complete. To do this the algorithm iterates n times, and at each vertex it must check each outgoing edge (m- rest of edges). While the algorithm has an embedded loop the time complexity is not quadratic; since the algorithm only looks at each edge once as all outgoing vertices are unique, the worst case is n+m time meaning the algorithm runs in O(n+m) time.

Oscar Rochanakij StuNum:1082645