COMP3821
Assignment 3

Charlie Bradford z5114682

May 21, 2018

1. (a) $A = \langle 1000, 1001, 1000 \rangle$
   (b) $A = \langle 1000, 1, 1, 1000 \rangle$

(c)
```
function GREATEST SUM(A, n)
    var m[n + 1]
    m[0] ← 0
    m[1] ← A[1]
    for i ∈ [1..n] do
        m[i] ← MAX(m[i − 2] + A[i], m[i − 1])
    return m[n]
```

2.
```
function MIN-COST RETREAT(CEO)
    m, n := MIN-COST RETREAT*(CEO) return MIN(m, n)
function MIN-COST RETREAT*(n)
    w ← n.cost
    o ← 0
    if n.next ≠ null then
        for x ∈ n.next do
            n, m ← MIN-COST RETREAT*(x)
            w ← w + n
            o ← o + m
    return o, w
```

3. This is a dynamic programming problem. Create a matrix that has the dimensions $(m + 1) * (n + 1)$, where m and n are the lengths of the two strings. Initialise the first column (the one of length $m + 1$) to 1 and the rest to 0. Move through the matrix, checking if each letter matches. If they do then the value is set as equal to the number of matches for all letter up to those currently considered for both strings, otherwise the value in the matrix is set to the same value for the next character in the longer string. This explanation was bad, but the psuedo code at the end is much better.

4. (a) This is a dynamic programming problem. Create an $(n+1)*(n+1)$ matrix and intialise every value to zero. Starting at 1,1, set every value to be the maximum of the values above and to the left plus the value stored in A. Full code at the end.

   (b) As above, but also intialise a $(n+1)*(n+1)$ matrix of empty strings. If the number above the square is larger append "D", else append "R", the string at n, n is the most valuable path.

   (c) As each value in the matrix depends on only its direct predecesor (either above or the the left) we can calculate the matrix in lines as below. The matrices below are just representations and the implementation could use arrays or some other method. xs represent unstored, empty values. The below shows path values, but an identical method could be put in place for storing paths. This method uses $O(n)$ extra space which

is less than the $O(n\sqrt{n})$ required.

$$\begin{bmatrix} x & 0 & x & x & x & \dots & x \\ 0 & x & x & x & x & \dots & x \\ x & x & x & x & x & \dots & x \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x & x & x & x & x & \dots & x \end{bmatrix}$$

$$\begin{bmatrix} x & 0 & 0 & x & x & \dots & x \\ 0 & A[1][1] & x & x & x & \dots & x \\ 0 & x & x & x & x & \dots & x \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x & x & x & x & x & \dots & x \end{bmatrix}$$

$$\begin{bmatrix} x & x & 0 & 0 & x & \dots & x \\ x & A[1][1] & A[1][1]+A[1][2] & x & x & \dots & x \\ 0 & A[1][1]+A[2][1] & x & x & x & \dots & x \\ 0 & x & x & x & x & \dots & x \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x & x & x & x & x & \dots & x \end{bmatrix}$$

$$\begin{bmatrix} x & x & x & 0 & 0 & \dots & x \\ x & x & A[1][1]+A[1][2] & A[1][1]+A[1][2]+A[1][3] & x & \dots & x \\ x & A[1][1]+A[2][1] & \max^{(A[1][1]+A[2][1]+A[2][2],}_{A[1][1]+A[1][2]+A[2][2])} & x & x & \dots & x \\ 0 & A[1][1]+A[2][1]+A[3][1] & x & x & x & \dots & x \\ 0 & x & x & x & x & \dots & x \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x & x & x & x & x & \dots & x \end{bmatrix}$$

$$\vdots \quad \vdots \quad \vdots$$

$$\begin{bmatrix} x & x & x & x & x & \dots & x & x \\ x & x & x & x & x & \dots & x & x \\ x & x & x & x & x & \dots & x & x \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x & x & x & x & x & \dots & x & v_1 \\ x & x & x & x & x & \dots & v_2 & x \end{bmatrix}$$

$$\begin{bmatrix} x & x & x & x & x & \dots & x & x \\ x & x & x & x & x & \dots & x & x \\ x & x & x & x & x & \dots & x & x \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x & x & x & x & x & \dots & x & v_1 \\ x & x & x & x & x & \dots & v_2 & \max^{(v_1+A[n][n],}_{v_2+A[n][n])} \end{bmatrix}$$

5.
- Add a super-source that connects to vertex u and the source s, and a super-sink that connect to vertex v and the sink t.
- Set the edges from the super-source to the s and u to have capacity equal to the gross output of the source and vertex u, respectively.
- Set th edges to the super-sink from t and the v have capacity equal to the gross input of the sink and vertex v, respectively.
- Use the Edmonds-Karp algorithm to find the max-flow.
- All edges accessible from the super-source in the residual flow network are one side of the cut.
- All other edges form the other side.

6. (a) The smallest amount of money T that allows her to go on all S rides is the amount with which she has the least leftover after receiving her final deposit back. Thus if T is the smalles amount of money then the last ride must be the one with the smallest deposit. Allowing for rides that may have deposits equal to or greather than the sum of the deposits for all rides with smaller deposits then those rides must also be riden before all the other rides, to allow for the smallest possible value of T. Thus if she has enough money to ride all S rides, then she has at least enough to ride them in non-increasing order of deposit.

(b) Sort D and C by their cost. For an array with $T+1$ values find the number of rides that can be gone on for each index in T by iterating through C and D to find the number and using the value for the previous index as a starting point.

(c) No, T may increase exponentially compared to n.

(d) Sort D and C by their cost. Iterate through C and D (low to high) and add on the next ride when possible to find the largest amount of rides to go on before the cost exceeds T. Sorting is $nlogn < n^2$ and there are at most n rides to go on.

---

function subStringCount(A, B)
  $lenA \leftarrow A.length$
  $lenB \leftarrow B.length$
  $p \leftarrow \{\{0\} * (lenA + 1)\} * (lenB + 1)$
  for $i \in [0..lenB$ **do**
    $p[0][i] \leftarrow 1$
  for $i \in [1..lenA]$ **do**
    for $j \in [1..lenB]$ **do**
      if $A[i-1] = B[j-1]$ **then**
        $p[i][j] \leftarrow p[i-1][j-1] + p[i-1][j]$
      else
        $p[i][j] \leftarrow p[i-1][j]$
  return $p[lenA][lenB]$

---

function Most Valuable Path(A, n)
  $mvp \leftarrow \{\{0\} * (n+1)\} * (n+1)$
  for $i \in [0..n]$ **do**
    $mvp[i][0] \leftarrow 0$
    $mvp[0][i] \leftarrow 0$
  for $i \in [1..n]$ **do**
    for $j \in [1..n]$ **do**
      $mvp[i][j] \leftarrow$ max($mvp[i-1][j]$, $mvp[i][j-1]$)
      $mvp[i][j] \leftarrow mvp[i][j] + A[i][j]$
  return $mvp[n][n]$

function Annotated Most Valuable Path(A, n)
  $mvp \leftarrow \{\{0\} * (n+1)\} * (n+1)$
  $ap \leftarrow \{\{\text{“”}\} * (n+1)\} * (n+1)$
  for $i \in [0..n]$ **do**
    for $j \in [0..n]$ **do**
      if $mvp[i-1][j] > mvp[i][j-1]$ **then**
        $mvp[i][j] \leftarrow mvp[i-1][j] + A[i][j]$
        $ap[i][j] \leftarrow$ Concatenate($ap[i-1][j]$, "D")
      else
        $mvp[i][j] \leftarrow mvp[i][j-1] + A[i][j]$
        $ap[i][j] \leftarrow$ Concatenate($ap[i][j-1]$, "R")
  return $mvp[n][n]$, $ap[n][n]$
