CS 180 Homework 4

March 8, 2015

1 Dynamic Coin Grabbing

(a) Show a sequence of $n \geq 6$ coins for which it is not optimal for the first player to start by picking up the available coin of larger value. That is, give an example for which the natural greedy strategy is suboptimal.

One such sequence of six coins in which the greedy approach fails is as follows:

1 2 3 4 1000 6

In this case, the coin grabbing goes as follows:

First Player	Second Player
6	1000
4	3
2	1

With this sequence of greedy grabbing, the first player ends up with a total of 12 while the second player has a total of 1004.

(b) Give an $O(n^2)$ algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute some information in $O(n^2)$ time, and then the first player should be able to make each move optimally in $O(1)$ time by looking up the precomputed information.

```
Function OptimalSequence(Array)
  Let MaxArray be a two-dimensional array of 0s
  Let A, B, and C be integers
  For every i from i to sizeof(Array)
    Let k be 0
    For every j from i to sizeof(Array)
      If k + 2 <= sizeof(Array - 1)
        A = MaxArray[k + 2][j]
      Else
        A = 0
      If k + 1 <= sizeof(Array - 1) And j - 1 >= 0
        A = MaxArray[k + 1][j - 1]
      Else
```

```
        A = 0
      If j - 2 >= 0
        A = MaxArray[k][j - 2]
      Else
        A = 0
      MaxArray[k][j] = max(Array[k] + min(A, B), Array[j] + min(B, C))
  Return MaxArray

Function TurnSequence(MaxArray, Array)
  Let A be 0, and Z be sizeof(MaxArray) - 1
  Let Turn be True
  Let PlayerMoves be an array
  While A < Z
    Let FromStart be MaxArray[A + 1][Z]
    Let FromEnd be MaxArray[A][Z - 1]
    If Turn = True
      If FromStart <= FromEnd
        Push Array[A] to PlayerMoves
        A++
      Else
        Push Array[Z] to PlayerMoves
        Z--
    Else
      Turn = !Turn
  Return PlayerMoves
```

(c) Find a sequence of $n \geq 6$ coins for which the dynamic programming strategy guarantees the first player a higher total value than the simple strategy

One such sequence of six coins in which the simple strategy gives a less optimal solution than the dynamic programming method is as follows:

3 2 2 3 1 2

In this case, the odd-numbered coins have a sum of 6, while the even-numbered coins have a sum of 7. Therefore, by taking only even coins, we end up with a total of 7. The dynamic programming solution however gives us a total of 8, which is higher than the simple strategy outcome.

2 Box Stacking

(a) Give an algorithm for box stacking that uses as few split and join steps as possible, when given two stacks of boxes which are represented as ascendingly sorted sequences of numbers (box sizes). Duplicate numbers are permitted in the input sequences, and the two stacks can contain identical numbers. Hint: the goal is to find the optimal split of one of two stacks.

```
Function BoxStack(First, Second)
  Let m be the number of boxes in First
  Let n be the number of boxes in Second
  Let P be a 2-d matrix of string arrays
  For i from 0 to m
    P[i][0] = "Join"
  For j from 0 to n
    P[0][j] = "Join"
  P[1][1] = "Join"

  For every i from 1 to m
    For every j from 1 to n
      If First[1] > Second[j] or First[i] < Second[1]
        P[i][j] = min(P[i-1][j] + "Join", P[i][j-1] + "Join")
      Else
        P[i][j] = min(P[i-1][j] + "Split", P[i][j-1] + "Split")

  Return P
```

(b) Determine the time complexity of your algorithm.

The time complexity is $O(n^2)$ at the worst case.

3 Longest Ascending Subsequence

(a) Give a Dynamic Programming recursion equation for $Len(x, i)$ in terms of $Len(x, 1)$, ... , $Len(x, (i-1))$.

$Len(x, i) = 1 + max(Len(x, k))$ for all k predecessors less than x_i

(b) Show how to use these recursion equations to obtain a polynomial-time algorithm for solving this problem.

The recursion can be used in an algorithm as follows:

```
For all i from 1 to n
  PreviousMax = 0
  If i != 0
    For all k from 0 to i - 1
      PreviousMax = max(PreviousMax, Len(x, k))
    Len(x, i) = 1 + PreviousMax
  Else
    Len(x, i) = 1
```

(c) What is the time complexity of this algorithm?.

The algorithm is $O(n^2)$ at its worst case.

4 Currency Exchange

(a) Because each entry in the matrix R is a rate, if we take the log of entries in the matrix, then sums of matrix entries are equal to logs of corresponding products of rates. Prove that the weight of a path using the log-transformed matrix $L = \log R$ is the log of the product of rates along that path. Also show the weight of the same path in the matrix $C = -L = -\log R$ corresponds to the inverse of the rate in L.

Let $A- > B- > C$ be a path in the original matrix R. Then the rate across this path is given the value $A * B * C$. The corresponding path in L is $\log A * B * C$, which can be reduced to the sum as follows: $\log A + \log B + \log C$. Furthermore the weights in the inverse matrix C will be $\log \frac{1}{A} + \log \frac{1}{B} + \log \frac{1}{C}$ which is reduced to $\log A^{-1} + \log B^{-1} + \log C^{-1}$, which further reduces to $-\log A + -\log B + -\log C$ and finally $-(\log A + \log B + \log C)$ which is clearly the negative of the corresponding path in L.

(b) Give an efficient algorithm (in pseudocode) to compute the shortest paths from each currency to each other currency using the matrix $C = -L = -\log R$.

Given an n x n matrix C the algorithm (Bellman-Ford) is as follows:

```
Function ShortestPath(Start)
  Let Distance be a matrix of n infinities
  Let Predecessor be a matrix of n -1s

  Distance[Start] = 0

  For every k from 0 to n
    For every i from 0 to n
      For every j from 0 to n
        If Distance[i] + C[i][j] < Distance[j]
          Distance[j] = Distance[i] + C[i][j]
          Predecessor[j] = i

  Return Distance
```

(c) The matrix C can have negative entries, and if we view it as the adjacency matrix of a graph, it can have negative cycles. Any negative cycle in the matrix C will correspond to a gain of money by currency trades and an opportunity for arbitrage. Give an efficient algorithm for finding the maximally negative cycle in the matrix C. (If there is no negative cycle, it should determine that.)

```
Function MaximalNegative(Distance)
  Let MaximumCycle be an empty array
  Let NegativeCycles be an empty array
  For every i from 0 to n
    For every j from 0 to n
      If Distance[i] + C[i][j] < Distance[j]
```

```
        Let N be the negative cycle around j
        Push N to NegativeCycles

If NegativeCycles is empty
  Return empty array

Let NegativeCyclesSums be an array of the sums of each NegativeCycle
Sort NegativeCyclesSums in decreasing order
MaximumCycle = NegativeCyclesSums[0] MaximumCycle

Return MaximumCycle
```