Homework 6 Solutions

Problem C-5.1

Algorithm planWalkabout(G, s, d, k):
Input: A graph G containing vertices s and d. Each vertex represents a watering hole. Vertex s and d represent the start and destination of Anatjari’s journey respectively. Each edge, e, in G contains a label with the distance between the endpoints of e. The value k is the maximum distance Anatjari can walk.
Output: The shortest path from s to d with no edges longer than k, if such a path exists.

for each edge e in G
  if the length of e is greater than k
    remove e from G
BFS(G, s)

if a path to d was found
  return the path
else
  indicate that no such path exists

Give a graph G with n vertices and m edges, the first for loop will run in O(m) time and the BFS algorithm will run in O(n + m) time. Therefore the algorithm will run in O(n + m) time.

The correctness of the algorithm rests on the fact that a BFS starting at a vertex s will find a path with the minimum number of edges (and hence vertices) between s and all other vertices of G (if such a path exists).

Problem C-5.5

Algorithm placeGuards(X, d):
Input: Sequence X of real numbers in sorted order where each value represents the location of a painting, and a distance d in which a guard can protect all paintings
Output: Sequence of real numbers, each value represents the location of a guard

\(^1\)See Theorem 6.19 in Goodrich-Tamassia, page 315.
G <- new empty sequence

if !X.isEmpty
    cur_pos <- X.elemAtRank(0) + d
    cur_area <- cur_pos + d
    G.insertLast(cur_pos)

    for i <- 0 to X.size()-1
        if cur_area < X.elemAtRank(i)
            cur_pos <- X.elemAtRank(i) + d
            cur_area <- cur_pos + d
            G.insertLast(cur_pos)

    return G

This algorithm uses the Greedy Method to find the minimum number of guards needed to protect the paintings in X. The running time of this algorithm is $O(n)$ where $n$ is the number of values in $X$.

Problem C-5.11

Like the Matrix Chain-Product problem, the problem of finding a minimum weight triangulation for a convex polygon can be solved using a dynamic programming algorithm.²

The algorithm below, minimumTriangulation(), fills in a two dimensional table, T, of values for a polygon P with $n$ vertices. A cell $T[i][j]$ stores the weight of the optimal triangulation of polygon $v_i, ..., v_j, v_{j+1}$ for $0 \leq i < j \leq n - 2$. Therefore cell $T[0][n-2]$ will store the weight of the optimal triangulation. By convention, the triangulation weight of a polygon $v_x, v_{x+1}$ is 0. The algorithm will run in $O(n^3)$ time and use $O(n^2)$ space.

Algorithm minimumTriangulation(P):
Input: Sequence P of n vertices, $v_{0}, v_{1}, ..., v_{n-1}$, representing a convex polygon
Output: An n-1 x n-1 table K where each cell $K[i][j]$ stores the index $k$ of the optimal weight triangle $v_{i}, v_{k+1}, v_{j+1}$

T <- an empty n-1 x n-1 table
K <- an empty n-1 x n-1 table
for i <- 0 to n-2
    T[i][i] <- 0

for d <- 1 to n-2
  for i <- 0 to n-d-2
    j <- i + d
    T[i][j] <- infinity
    for k <- i to j-1
      q <- T[i][k] + T[k+1][j] + diagonal_weight(P.elemAtRank(i),
                                           P.elemAtRank(k+1),
                                           P.elemAtRank(j+1))
      if q < T[i][j]
        T[i][j] <- q
        K[i][j] <- k
  return K

Given the index table produced above, the constructTriangulation() algorithm reconstructs the optimal triangulation of the polygon P.

Algorithm constructTriangulation(P, K, i, j):
Input: An n-1 x n-1 table K where each cell K[i][j] stores the index k of the optimal weight triangle v_{i}, v_{k+1}, v_{j+1}.
Sequence P of n vertices, v_{0}, v_{1}, ..., v_{n-1}, representing a convex polygon. Indices i and j
Output: Sequence of triangles used to triangulate P
if j > i
  k <- K[i][j]
  X <- constructTriangulation(P,K,i,k)
  Y <- constructTriangulation(P,K,k+1,j)
  Z <- merge(X,Y)
  Z.insert(new triangle(P.elemAtRank(i),
                        P.elemAtRank(k+1),
                        P.elemAtRank(j+1))
  return Z
else
  return an empty sequence

An n vertex polygon may be triangulated using exactly n - 3 triangles. Therefore the constructTriangulation() method runs in O(n) time.

The computeTriangulation() method is a wrapper around the above to methodds that provides a convienent interface to an external user.

Algorithm computeTriangulation(P):
Input: Sequence P of n vertices, v_{0}, v_{1}, ..., v_{n-1}, representing a convex polygon.
Output: The set of triangles that form the minimum weight triangulation of $P$

$$K \leftarrow \text{minimumTriangulation}(P)$$
return $\text{constructTriangulation}(P, K, 0, n-2)$

**Problem R-6.8**

Given a graph $G$ implemented using the adjacency matrix structure with $n$ vertices, the method $\text{incidentEdges()}$ will need to examine $n$ cells to determine the number of edges incident on a particular vertex. Therefore the running time of $\text{incidentEdges()}$ is $\Theta(n)$.

Notice that the DFS algorithm is called exactly once for each vertex in $G$. Each execution of the DFS algorithm results in one invocation of $\text{incidentEdges()}$ and the running time of $\text{incidentEdges()}$ dominates the running time of DFS as a whole. Therefore the total running time of the algorithm is $\Theta(n^2)$. 
Program

/*
 * Implementation of the dynamic programming algorithm for finding the
 * longest common subsequence of two strings.
 *
 * @author James Lentini
 */

public class LCS
{
    public static String lcs(String x, String y)
    {
        int L[][] = new int[x.length() + 1][y.length() + 1];
        StringBuffer lcs = new StringBuffer();

        // initialize L
        for ( int i = 0; i <= x.length(); i++ ) { L[i][0] = 0; }
        for ( int j = 1; j <= y.length(); j++ ) { L[0][j] = 0; }

        // fill in L
        for ( int i = 1; i <= x.length(); i++ )
        {
            for ( int j = 1; j <= y.length(); j++ )
            {
                if ( x.charAt(i-1) == y.charAt(j-1) )
                {
                    L[i][j] = L[i-1][j-1] + 1;
                }
                else
                {
                    L[i][j] = Math.max(L[i-1][j], L[i][j-1]);
                }
            }
        }

        // reconstruct the lcs
        int i = x.length();
        int j = y.length();

        while ( i != 0 && j != 0 )
        {
        }
if ( x.charAt(i-1) == y.charAt(j-1) )
{
    lcs.insert(0, x.charAt(i-1));
    i--;
    j--;
}
else if ( L[i-1][j] > L[i][j-1] ) { i--; }
else { j--; }
}
return lcs.toString();
}

public static void main(String[] args)
{
    String x = "James", y = "Lentini";
    System.out.println("An LCS of " + x + " and " + y + " is " + lcs(x, y));
}