1. m = 18

π =

0	0	1	1	2	3	4	5	6	7	8	9	2	3	4	5	6	7

2. Only the KMP-MATCHER algorithm will need to be changed to accommodate this modification. The COMPUTE-PREFIX-FUNCTION(P) algorithm will be the same. The following is the modified KMP-MATCHER algorithm:

```
KMP-MATCHER (T,P)
        n = T.length
        m = P.length
        q = 0
        maxQ = 0                        // max number of prefix chars matched
        shift = 0                       // shift for prefix
        for (i = 1) to n
                while q > 0 and P[q+1] != T[i]
                        q = π[q]

                if P[q+1] == T[i]

                        q++

                if q == m

                        maxQ = m

                        q = π[q]

                        shift = i-m

                if q > maxQ

                        maxQ = q

                        shift = i-q

                        print “P’s largest prefix occurs at substring index “ shift
```

3. $X = (x_1, x_2,...,x_M)$ $Y = (y_1, y_2,...,y_N)$
   PRINT-LCS(*c,X,Y,I,j*) should accept as a parameter *c* containing completed values of the c table and print out the LCS of the original sequences X and Y. Since it cannot use the b table, it will only compute the entries of b when needed. Since variables i and j or both are decremented by 1 after each iteration of the while loop until they reach 0, the time complexity of the algorithm is O(m+n). The algorithm is as follows:

```
PRINT-LCS(c,X,Y,I,j)
if c[i, j] == 0
        return
if X[i] == Y[j]
        PRINT-LCS(c,X,Y,i-1,j-1)
        print X[i]
elseif c[i-1,j] > c[I,j-1]
        PRINT-LCS(c,X,Y,i-1,j)
else
        PRINT-LCS(c,X,Y,i,j-1)
```

4. The pseudo-code for my algorithm solution is as follows:

```
Knapsack(n,W)
        Initialize a table K of size n+1 by W+1
        for y = 1 to W
                K[0, y] = 0
        for x = 1 to n
                K[x, 0] = 0
        for i = 1 to n
                for j = 1 to W
                        if j < i's weight then K[i, j] = K[i-1,j]
                K[i,j] = max(K[i-1, j], K[i-1,j-i.weight] + i.value)
```

   Pick the lightest and most valuable item.

Proof of argument: If there were an item that we included, but a valuable and smaller item i existed, we could swap item j in the knapsack with the item i. This would increase the total value because the item i is more valuable and it will also fit because i is lighter than j.

5. We may use a ID array arr to solve the variation of the problem, where arr[W+1] can be used such that arr[i] stores the maximum using all items and i capacity of the knapsack. The algorithm is as follows:

```
infiniteKnapsack(int W, int n, int[] val, int[] weight)
        int arr[] = new int[W+1]
        from i = 0 to W
                for j = 0 to n
                        if weight[j] <= i
                                arr[i] = max(arr[i], arr[i-weight[j]] + val[j]
        return arr[W]
```

6. A maximum spanning tree can be produced by modifying the minimum spanning tree algorithm to negate all edge weights and applying the MST algorithm rule, multiplying -1 to all edge weights, and then applying Kruskal's or Prim's algorithm to find the minimum spanning tree.

```
Let,

        setOfVert = Set of vertices

        setOfEdges = Set of edges

Return maximumTree(setOfVert, setOfEdges)

for each edge in setOfEdges

        edge.weight = (-1) * edge.weight                    // sets all edge weights to negative

MST = Kruskals(setOfVert, setOfEdges)

maximum = setOfEdges

for each edge in MST.edges

        maximum.remove(edge)

Return maximum
```

7. Dijkstra's algorithm marks vertices as "closed" under the assumption that any node originating from it will lead to a greater distance. Thus, the algorithm finds the shortest path to it and never develops this node again. This does not hold true in the case of negative weights though, as shown below:

   A -> C = 2
   A -> B = 5
   B -> C -6

8. We may prove that the all-pair-shortest-path algorithm is correct by giving an example. Consider Floyd Warshall's algorithm with the following edges:

   1 -> 2 = 4

   2 -> 3 = 3

   3 -> 1 = -5

   This graph contains a negative edge but does not contain a negative cycle.

   The distances are as follows:

   (1,1) = 0

   (1,2) = 4

   (1,3) = 7

   (2,2) = 0

   (2,3) = 3

   (3,1) = -5

   (3,2) = -1

   (3,3) = 0

   Therefore, we can see that the algorithm is correct in a graph given a negative edge with no negative cycles.

9. The algorithm goes as follows:

   Apply the Floyd Warshall algorithm on the graph G.

   min = ∞

   for each vertice (u,v)
   ```
       if (dist(u,v) + dist(v,u) < min)
           min = dist(u,v) + dist(v,u)
           pair = (u,v)
           return path(u,v) + path(v,u)
   ```