AOA Assignment 4

Sunny Shah - 112044068

November 7, 2018

1) Page 312 problem #3

a)

The greedy algorithm described do not work for the following graph:

Nodes: 1, 2, 3, 4, 5

Edges: (1 -> 2), (1 -> 3), (1 -> 4), (2 -> 5), (3 -> 4), (3 -> 5)

The greedy algorithm discussed gives the answer as 2 i.e. (1 -> 2), (2 -> 5)
However, the correct answer should be 3 i.e. (1 -> 3), (3 -> 4), (4 -> 5)

b)

Since, the graph has directed edges such that, each edge starts from small indexed node to higher indexed node, we can compute the max distance to each node using dynamic programming approach as follows:

```
Let dp[i] => Max distance to node i from node 1.

# base case:
dp[1] = 0

# Initialize max distance to other nodes as -1, denoting that they cannot be visited
dp[2,3...n] = -1

for i -> 1...n:
    # if the current node is accessible, then only update the paths of other nodes
    if dp[i] != -1:
        # Using the current node, update all nodes directly accessible from node i
        for each edge ( i,j ):
            dp[j] = max( dp[j] , 1 + dp[i] )

return dp[n]
```

Complexity:

The above algorithm evaluates each edge in the graph once and the total edges in the graph described can be $\frac{n*(n-1)}{2}$, where n is the number of vertices.
Thus, the overall complexity of the above algorithm is $O(n^2)$

2) Page 312 problem #5

Solution:

- We can use the following logic to solve this problem:

  If we are considering the characters from index i to j as being a single word, then we can find the optimal quality of string till j as being equal to ($quality\_of\_substring(i, j) + optimal\_quality\_of\_substring(0, i-1)$)

- Thus, we can formulate our dp as follows:

  dp[i] = $\forall(j <= i)$ max(quality(substring(j , i)) + dp[j - 1])

Algorithm:

```
Let dp[i] => Max quality attained for the prefix of length i for the string.

# null string, since we consider the string to have start index as 1
dp[0] = 0

Let n be the length of the string.

# Let substring(i,j) be a function returning
# substring from index i(inclusive) to index j(inclusive)

for i -> 1..n:
    for j -> i , i-1 , ...1:
        dp[i] = max( dp[i] , quality( substring(j,i) ) + dp[j-1] )

return dp[n]
```

Complexity:

For evaluation of dp[i], we need to look for i previous subproblems, thus, the overall complexity of the algorithm is $O(n^2)$

3) Page 312 problem #13

Solution:

- The solution requires us to find a cycle such that the product of ratios along the cycle is greater than 1. i.e. r1 * r2 * r3 * ... > 1
- We can decompose the problem a little by taking log on both the sides:
  log(r1) + log(r2) + log(r3) ... > 0

  Thus, all we need to do is to find a cycle in graph, such that addition of log of each ratio is greater than 0.
- Let take negate the above equation, that gives us: -log(r1) - log(r2) - log(r3) .... < 0
- Now, if we plot a graph with each share as a single node and the negative log of ratio(r_{ij}) i.e. $-log(r_{ij})$ between share i and j as the weight of edge between i and j, the problem then reduces to finding a negative cycle in the graph.
- Finding the presence of a negative cycle in a graph can be easily accomplished using Bellman-Ford algorithm

Algorithm:

```
Let us denote the edges in the graph using an adjacency matrix weight:

for each (i,j):
    Let weight[i][j] = -log(ratio_ij)

# We use the distance array that stores the distance of node i from the source node.
# Initialize the distance array as infinity for all the nodes except the source node.
Let distance[1..n] = infinity

# Define a parent array that stores the index of the node
# via which the shortest path was found. Initialize to -1.
parent[1...n] = -1

# Consider the node 1 as the source node
distance[1] = 0

for i -> 1...n-1:
   for each edge (u, v) with edges:
       if distance[u] + weight[u][v] < distance[v]:
            distance[v] = distance[u] + weight[u][v]
            parent[v] = u

negative_cycle_node = -1
# Find the negative weight cycle in the graph
for each edge (u, v) in edges:
   if distance[u] + weight[u][v] < distance[v]:
       negative_cycle_node = v
       break

if negative_cycle_node != -1:
    print the negative cycle using the parent array.
else:
    print "No opportunity cycle found!"
```

4) Page 312 problem #19

Solution:

- Let us try to develop the recurrence relation.
- Let dp[i][j] denote that there is a way to partition into two subsequence a,b such that a is first i characters of string x^k and b is first j characters of string y^k
- Now, we can see that dp[i][j], is possible if:

  dp[i-1][j] is true and the $(i+j)^{th}$ character is equal to i^th character of x^k
  OR
  dp[i][j-1] is true and the $(i+j)^{th}$ character is equal to j^th character of y^k
- Thus, our recurrence relation can be denoted as:

$$dp[i][j] = \begin{cases} true & \text{if (dp[i-1][j] AND s[i+j] == } x^k\text{[i])} \\ true & \text{if (dp[i][j-1] AND s[i+j] == } y^k\text{[j])} \\ false & otherwise \end{cases}$$

Algorithm:

```
# Initialize dp[ 1...n ][ 1...n ]
dp[1...n][1...n]

#Create a string of x^k and y^k of length n
x_k = x*x*x...[:n]
y_k = y*y*y*..[:n]

# base case
dp[0][0] = true

for i -> 2...n:
    for i -> 1...n:
        j = n - i
        if( dp[ i-1 ][ j ] and s[ n ] = x_k[ i ] OR
            dp[ i ][ j-1 ] and s[ n ] = y_k[ j ]  )

            dp[ i ][ j ] = true

for i -> 1...n-1
    j = n - i
    if dp[ i ][ j ]
        return "String s is interleaving of x and y"
return "s is not an interleaving of x and y"
```

5) Page 312 problem #24

Solution:

- Total number of voters from n precincts, each of which has m voters is m*n
- For party A to dominate in each district, it will need to have at least $\frac{n*m}{4}+1$ votes in each district, as each district will have $\frac{n*m}{2}$ votes and party should have majority in that district.
- Thus, party A should have overall, at least $\frac{m*n}{2}+2$ votes to dominate in both district, with at least $\frac{n*m}{4}+1$ votes in each district.
- Thus, we need to find a district n/2 precincts, such that, number of votes V in such a district is:

  $\frac{n*m}{4}+1 \leq V$
- Also, the votes in other district has to dominate for party A.

  Let A_total be total votes for party A in all precincts, then

  A_total - $(\frac{n*m}{4}+1) \geq (\frac{n*m}{4}+1)$

  Thus, we need to find a V such that,

  $(\frac{n*m}{4}+1) \leq V \leq A\_total - (\frac{n*m}{4}+1)$
- The above equations assumes that the party A leads to gerrymandering, however, what if party B can give rise to gerrymandering.
- To find which party can give rise to gerrymandering, we can count the number of votes of party A and party B. If number of votes of B is more than that of A, then party can maybe dominate in both district. But, to make the problem easy, we swap the votes of A and B in all precincts, if B is majority. Thus, we now only need to check if A can lead to gerrymandering.
- Let us formulate our DP:
  Let dp[i][j][k] denote, that out of i initial precincts, there are j precincts we can choose, such that sum of votes for party A is k.
  Thus, dp can be formulated as the following recurrence relation:

$$dp[i][j][k] = \begin{cases} true & \text{if dp[i-1][j-1][k-A[i]] == true} \\ true & \text{if dp[i-1][j][k] == true} \\ true & \text{if j=0 and k=0} \\ false & otherwise \end{cases}$$

  where, A[i] => number of votes for party A in i^{th} precinct

Algorithm:

```
# initially all false
# i -> 0 to n
# j -> 0 to n/2
# k -> 0 to A_total - ((n*m)/4 + 1)
dp[i][j][k]

dp[0][0][0] = true

# Votes for A in each precinct:
A[1...n]

# Votes for B in each precinct:
B[1...n]

# Sum of votes for party A and B
A_total, B_total

if A_total < B_total:
    swap A[1...n] and B[1...n]
    swap A_total and B_total

for i -> 1...n:
    # it is always possible to choose 0 precinct to bring 0 votes for A.
    dp[ i ][ 0 ][ 0 ] = true

    for j -> 1...n/2:
        for k -> 1...(A_total - ( (n*m)/4 +1 )):
            if ( k-A[i] >= 0
                    and k-A[i] <= (A_total - ( (n*m)/4 +1 ))
                        and dp[ i-1 ][ j-1 ][ k-A[i] ] == true ):
                    dp[ i ][ j ][ k ] = true

            else if ( dp[ i-1 ][ j ][ k ] == true ):
                    dp[ i ][ j ][ k ] = true

            else
                    dp[ i ][ j ][ k ] = false

# Now, we just need to find if it is possible to have n/2 precincts out of n precincts,
# such that the number of votes to A is between
# (n*m)/4+1 and ( A_total - ((n*m)/4+1) )

for i -> ( (n*m)/4 +1 ) .... (A_total - ( (n*m)/4 +1 ) ):
    if dp[ n ][ n/2 ][ i ] == true:
        return true

return false
```

Complexity

From the algorithm, it is clear that the loops run for n * n * (A_total - (n*m)/4 +1), which is same as n*n*(n*m) as A_total can be max n*m. Thus, the complexity of the above algorithm is $O(n * n * n * m) \equiv O(n^3 m)$