Project Submission 2
Sahil Palnitkar
CSCE 411

4)
a) Considering an isolated column, we can have 8 placements by placing 0, 1 or 2 pebbles.
{},{1},{2},{3},{4},{1,3},{1,4},{2,4}

b) Make an array dp[n][5][5]. make array temp[maxn][5].
We assume that j<k for generality.

```
algorithm pebble(i,j,k):
if i == n
    return 0
ans = dp[i][j][k]
if ans != -1
    return ans
ans = pebble(i+1,0,0)
for x from 1 to 4:
    if x != j and x != k:
        ans = max(ans, a[i][x] + pebble(i+1,0,x)
if j != 1 and k != 3:
    ans = max(ans, a[i][1] + a[i][3] + pebble(i+1, 1, 3))
if j != 1 and k != 4:
    ans = max(ans, a[i][1] + a[i][4] + pebble(i+1, 1, 4))
if j != 1 and k != 4:
    ans = max(ans, a[i][2] + a[i][4] + pebble(i+1, 2, 4))
return ans

algorithm checkerboard:
    for i from 0 to n:
        for j from 1 to 4:
            ans = pebble(0,0,0) //to get the max value
```

Correctness:
Let dp[i][j][k] store the max sum that can be achieved starting from column ‘I’ to column ‘n-1’ (columns are numbered 0 to n-1, where row j and row k of column i-1 are chosen). If j=0 or j=0, then no cell is chosen. rows are numbered 1 to 4.) When the algorithm checkerboard runs, it calls pebble with (0,0,0) to get the max sum value.

Time Complexity:
Checkerboard runs a for loop n times giving O(n) complexity. The other for loop that runs inside the original for loop and the loop that runs inside pebble all run with constant time scaling. This brings the final time complexity to O(n)

7)
We can define the subproblems as
for $1 \leq i \leq X$ and $1 \leq j \leq Y$ let C(i,j) be the best return that can be obtained from a cloth of shape ixj.
let function rectangle be :
rectangle(i,j) = maxk ck for all products k with ak = I and bk = j
or 0 if no product exists
Implementation:
$C(i,j) = \max\{\max_{1\leq k<i} \{C(k,j) + C(i-k, j)\}, \max_{1\leq h<j} \{C(i,h) + C(i, j – h)\}, \text{rectangle}(i, j)\}$
We can initialize the subproblems as C(1, j) = max{0,rect(1,j)}
and C(i, 1) = max{0,rect(i,1)}
The final solution will be the value of C(X,Y)

Correctness:
We can prove correctness by seeing that C(i,j) trivially has the intended meaning for the base cases when i = 1 or j =1. Inductively, C(i,j) is solved correctlym as a rectangle ixj can only be cut in the (i-1) + (j-1) ways considered by the recursion or be occupied completely by a product, which is accounted for by the rectangle(i,j) term.

Running Time:
The total running time O(XY(X+Y+n)) as there are XY subproblems and each of them take O(X+Y+n) to solve.

8)
the maximum sucesss probability would be the minimum failed probability, and each machine should be at least 1, so we can reduce the original package B into B'= B-sum(ci), and each machine's dp[i][0] should be initialized as fail probability (1 - ri).
Using a modified knapsack problem
dp[i][c] = min(dp[i-1][c], dp[i-1][c-cost[n]]*(1-ri))
0<= c <=B' , 1<=i<=n, B' = B-sum(ci)

we can initialize arrays like:
dp[0][0~B'] = 1
dp[i][0]   = 1-r_i, 1<=i<=n

Algorithm:
```
for(int i = 1 ; i<=n;++i)
  for(int c = cost[i] ; c<=B';++c)
     dp[i][c] = min(dp[i-1][c], dp[i-1][c-cost[n]]*(1-ri)))
```

Correctness:
This is a modified form of the knapsack problem so we know that this will consider the reliability as the weights for all the steps i along the system.

12)
Let G = (V,E) be the following graph: G has a node corresponding to every variable xi , and there is an edge (xi , xj) if xi = xj is an equality constraint.

We do a Depth First Search of G to find all its connected components. For each node xi , if it is in the k-th connected component of G, we assign it a mark k.
Now for each inequality constraint xi != xj , check to see if xi and xj occur in different connected components of G.
If there is some inequality constraint xi != xj such that xi and xj occur in the same connected component of G, then we report that the constraints are unsatisfiable. If there is no such constraint, then we report satisfiable.

Correctness: The equality constraints form an equivalence class. If all the constraints are satisfiable, then the inequality constraint cannot exists. If xi and xj are in the same equivalence class, this means that such a constraint is not satisfiable, and this is a contradiction. So, if all the constraints are satisfiable, the algorithm works.
If all constraints are not satisfiable this is only possible if there is an inequality constant xi != xj where xi and xj are in the same equivalence class. If this is not true, and xi and xj are always in the different connected components. So an assignment that leads to a k mark to all the variables in a connected component k satisfies all constraints, therefore this is a contradiction. So we can say that if the constraints are not satisfiable, the algorithm is still correct.

Time Complexity: We can create the result set by going through the connected list and marking components. Giving us time complexity of O(n)

13)
a) Let (d1,d2,d3,d4) = (3,3,1,1). The sum is 8 and di $\leq$ 3. No graph exists with this degree sequence.
b)
i. We know that v1 has d1 neighbors and they are not v2, v3, …, vd1+1, then (v1,vj) in E for some j with d1 +1 < j $\leq$ n and (v1,vi) not in E for some i with 1 < i $\leq$ d1+1. Which implies i < j $\leq$ n. We know that vj has dj neighbors and vi has di neighbors. As vj is a neighbor of v, dj > 0. The ordering of degrees implies that di $\geq$ dj as we know i < j. The neighbors of vj include v1 and the neighbors of vi do not so di $\geq$ dj implies that there must be a vertex u != vj that is a neighbor of vi and not vj. Therefore (u, vi) in E and (u,vj) not in E.

ii. Find a vi, vj and u like we did in part i. These vertices exist as we proved in part i. Let E0 = E OR {(v1, vi),(u,vj)} \ {(v1, vj),(u,vi)}. For vertices v1,vi,vj,u in V, the degrees are the same. Therefore G0 = (V,E0) has the same degree sequence and (v1,vi) is an edge.

iii. We can repeat the above procedure until the neighbors of v1 are v2, … , vd1+1. This sequence will always terminate because d1 is finite.

c)
Let d1,…dn be the desired sequence. To decide if a graph exists with this sequence, we can use the following procedure. If the sequence has length 1, output true if d1 = 0 else output false. If the length is not 1, sort the sequence into d1’ $\geq$ ... $\geq$ dn’. If d1’ < 0, or if d1’ $\geq$ n, return false. Else, remove d1’ from the sequence , subtract 1 from (d2’, …, $d'_{d1'+1}$), do the following operations and then continue.

Implementation: Store the sequence d1 ≥ d2.... ≥ dn as a sorted linked list D. Separately store two other sorted linked lists First and Last. First(i) and Last(i) represent the first and last occurrence of degree i in the linked list D; these two lists are maintained in decreasing order of degree.

In each recursive step, the head of the linked list D will be deleted and the first few elements in the list will be decremented by 1. There are two cases-

In the first case, this operation will not disrupt the sorted order of the linked list D. If the head had degree dmax before the operation, updating D as well as the First and Last lists will take O(dmax) time, and thus the total update operation will take O(dmax) time.

In the second case, the sorted order of the linked list D will be disrupted. However, we can take advantage of the fact that the first few elements of D are decremented by exactly 1. Suppose after the decrement operation, dj , . . . , di becomes smaller than di+1. The observation is that they become smaller by exactly 1; so we can use the Last array to determine the last node with degree di+1 in D and insert dj , . . . , di after this last node. This insertion will take constant time. We'll need to update First and Last, which again takes O(dmax) time, where the head had degree dmax before the decrement.

Correctness: The proof of correctness is by induction on n. Suppose n = 1; there are no other nodes to connect to, so d1 must be 0 to make a graph. Now, suppose that the algorithm is correct for n = k, and we will show that it is also correct for n = k + 1. Let d1, . . . , dk+1 be the desired degree sequence and d1' ≥ . . . ≥ d'k+1 be the sequence sorted in nondecreasing order. The previous part showed that if a graph exists with this sequence, then there is a graph with this sequence that also has {v2, . . . , $v_{d1'+1}$} as the neighbors of v1. Such a graph, if v1 is removed, has k vertices and degree sequence given by d2' − 1, . . . , $d_{d1'+1}$-1, $d_{d1'}$, . . . , d'n . The induction hypothesis implies that the existence of such a graph is correctly determined by the algorithm. If such a graph exists, then the k + 1-vertex graph exists as well by attaching the new vertex v1 to v2, . . . , vd'1+1.

Running time:
The original sequence can be sorted in O(nlogn) time. There is at most one recursive call corresponding to each di in the original sequence and decrements are performed. The linked lists can be rearranged and updated in O(d1') operations. The total running time comes to O(nlogn + m) where m = $\sum^{i=n}_{i=1}$di

14)
We can build a tree for each leaf node for a unique character for all leaf nodes. The least frequent character will be at root.

Algorithm:
Join the pair of trees with the smallest frequencies till we get to the root node.
Repeat these steps until all trees only have one node which is the root node.

Correctness:
Each tree contains only one node, which corresponds to each letter. We can check the tree and traverse it to write the code word. The cost of this encoding will be m$\sum_{i=1}$ fi x di. Here fi is the frequency of the ith word and di is the depth of the ith word. This will work as all the characters will be processed.

Running Time:
This will run in O(mlogm) time.

15)
A greedy algorithm will help the customer with the minimum time taken to serve first. Which means that the customers will be sorted by their t values and served in that order. The total time taken to serve all customers will not change. If we want fewer customers waiting, we will serve customers by increased service time.

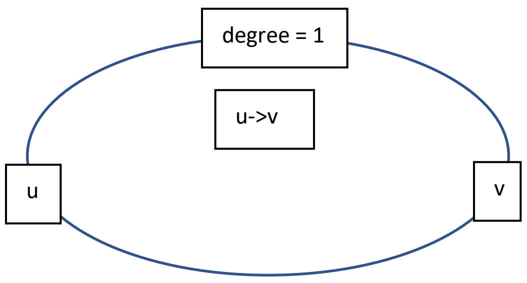
Algorithm:
Here input is the customers n, and the set of time for each customer T.
Output is the optimal service order

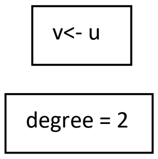
```
optimal_service(n, T):
sort T in ascending order of time ti.
return T.
```

Correctness:
Assume that there is an optimal solution that is better than this greedy solution. This will mean that the optimal solution will contain a pair of customers where service time for customer 1 is higher than that of customer 2.
This means that t1 > t1+1
If we swap t1 and t2, customer 2 will have to wait less. This swap will not affect the waiting time of other customers. Each swap of this out of order waiting time, will reduce the total waiting time.
This means that the greedy algorithm must be the optimal one.

Running Time:
As we are sorting a list of integers, if we use quicksort, we get running time of O(nlogn).

16)
a) It is important to node that in sum for u in V d(u), each edge is counted twice, one from degree d(u) and another from degree d(v)





In the above graph, the vertices v and u contribute two degrees to traverse paths v to u and u to v. Eatch edge adds 2 to the sum d(b). An edge has two vertices and they are counted twice. Therefore we get the relation sum u in V d(u) = 2|E|.

b) Assume that Vo represents the set of vertices with odd degree and Ve represents the set of vertices with even degree. Then consider

$\sum_{u \text{ in } V} d(u) = \sum_{u \text{ in } Ve} d(u) + \sum_{u \text{ in } Vo} d(u)$

From part (a)

$\sum_{u \text{ in } V} d(u) = 2|E|$

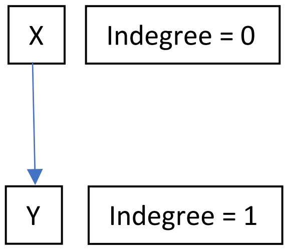
From the previous equations

$\sum_{u \text{ in } Ve} d(u) + \sum_{u \text{ in } Vo} d(u) = 2|E|$

$\sum_{u \text{ in } Vo} d(u) = 2|E| - \sum_{u \text{ in } Ve} d(u)$

the RHS is even. The LHS will be the sum of odd numbers. The LHS can be even only if it is the sum of an even number of odd numbers. So, there must be an even number of vertices in Vo.
So, it is proved that in an undirected graph, there must be an even number of vertices whose degree is odd.

c) We cannot prove that there are an even number of vertices with an odd degree.



From this graph, vertex X has indegree 0 and vertex Y has indegree 1. There is only one vertex with an odd degree (vertex Y). Therefore we cannot prove that the directed graph has an even number of vertices.

17)
white node = node is unvisited,
gray = node is partially visited //all routes not explored.
black = node is completely visited

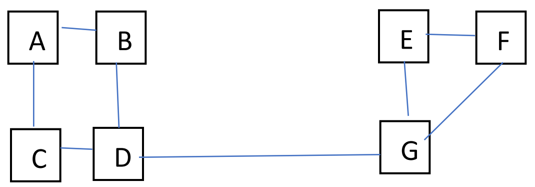
A graph G(V,E) has a cycle containing edge e {u,v} iff there is a path from u to v after removing an edge e from the graph. It can be made in linear time O(|V| + |E|) using DFS.

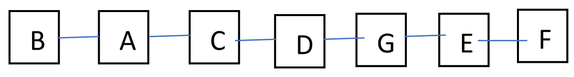
```
traverse(G-e):
for all v in V:
    color(v) = white
DFS(u)

DFS(u):
color(u) = gray
for each v in adjacent[u]:
    if color(v) = white:
        DFS(v)
color(u) = black
//the node is fully visited
```

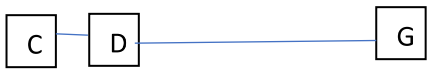
Correctness:
The algorithm goes through each node and marks it as unvisited. Then going through the nodes one by one, marks them gray if visited once. If all the nodes adjacent to the current node are visited, the node becomes black. In this way, all the vertices are visited one after the other to determine whether the graph has a cycle.

Running Time:
The time taken to visit all nodes is |V|. Each edge will be traversed twice in |E| time. This makes the total runtime O(|V|+|E|) which is linear.

18)
a)Considering a connected undirected graph G with a set of vertices V. If a graph is connected, it means that there is a path connecting two vertices. We know that the graph G is connected, so DFS will give a single connected component on running DFS. we can run DFS from any vertex on G to produce a DFS tree. If we remove v from the bottom of this tree, the result will still be connected. The result will be the connected subgraph of the new graph obtained by removing v.
Therefore, it is proved that there exists a vertex v in V in a connected undirected graph G = (V,E) such that the graph is still connected even though v is removed.



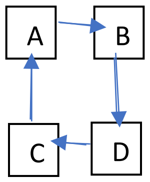




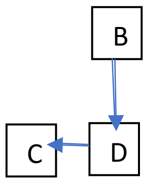
After removing vertex F, the resulting DFS tree is still connected. Therefore the final graph is also connected.

b) Consider a graph with a directed cycle. A directed cycle is always strongly connected. But if we remove a vertex from the cycle, the resultant subgraph is not strongly connected.

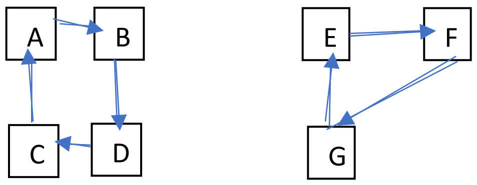
Consider this cycle.



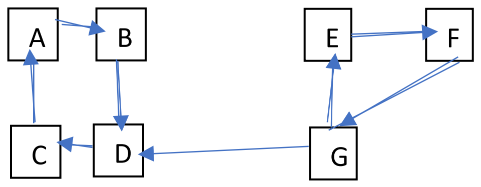
The above graph is strongly connected as there is a path between any two nodes. If we remove vertex A, the graph will not be strongly connected.



c) Considering a directed graph consisting of two disjoint cycles. Since a directed cycle is always strongly connected, these two disjoint cycles are strongly connected. To make this graph strongly connected, we need to add at least two more edges. One going from component 1 to component 2 and one from component 2 to component 1.
Therefore, we cannot make the graph strongly connected by adding just one edge.



If we add an edge from G to D, we get



Which is not strongly connected.

19)
We can implement a linear time algorithm which takes a DAG and two vertices s and t in G and return the number of simple paths between the vertices can be implemented by modifying DFS.

We can run DFS starting at s and ending at t, marking t as black when it is discovered. Set the count of vertices to the sum of counts of all vertices adjacent to the one most recently processed. Return number of paths.
Let every vertex have a variable 'paths' that counts paths leading out of it.

```
Algorithm simple_path(s,t):
if s ==t:
    return 1
else
    for w in adjacent[s]:
        s.path = s.path + simple_path(w,t)
    return s.path
```

Correctness:
The algorithm returns 1 if the start and end nodes are the same. Otherwise, the function runs recursively to count the number of paths starting at s, going to nodes adjacent to s and ending at t. This does not add partially completed paths.

Running Time:
As the function runs recursively for each vertex and edge in the graph, it is in linear time O(V+E)

20)
If a path exists that touches each vertex only once, then there is an edge between any two nodes in the topological sort of G.

To find if such a path exists we can:
Linearize the DAG G.
Take every two consecutive noes in the linearized order of G and check whether there is an edge between those two nodes. If there is a node between all two consecutive edges, then such a path exists. Else it doesn't.

We can linearize using DFS:
We will run DFS on the DAG G.
Vertex with highest post number is the source, which means its indegree is zero. We will remove the source from the graph.

We will find the next source vertex in this new graph.
Repeat till all vertices are processed.
The order in which we remove vertices gives us the linearized order of the graph.

```
Algorithm path(G):
DFS(G)
add initial source vertices found by DFS to list L.
while (L !empty):
    remove v from L
    add v to linear order
    reduce indegree of neighbors of v by 1 //this will remove v from the graph
    for vertex in neighbor[v]:
        if vertex.indegree == 0:
            add vertex to L
for (vertex, vertex+1) in linear order:
    if edge between (vertex,vertex+1) does not exist:
        return false
return true
```

Correctness:
The algorithm starts at the source vertex and removes it. This process is repeated for the resultant graph. After all vertices get processed, we go to the linear order and check if each adjacent vertex has an edge between it. If it doesn't, we return false as no path exists that touches each vertex only once. If there is, this path exists and we return true.

Running Time:
Linearization uses DFS and removes vertices recursively.
Therefore this takes $O(|V|+|E|)$ time which is linear in the size of the graph.
So the total runtime is linear.

21)
We already know that the number of vertices with an odd degree in an undirected graph must be even. Suppose the number of such vertices is 2k. We can randomly pair these vertices and add an edge between each pair to make all vertices of even degree.
We know that each connected component of this new graph will have an Euler tour. We will remove the k edges we added from this set of tours and break it into k paths with the two nodes at the end of each path being of odd degree. Also, all these paths will be edge disjoint as an Euler tour uses each edge exactly once. So, if we take the two ends of each path as a pair, we will get the necessary pairing.

22)
If T is a shortest path tree, then for each edge e = (u,v) in E-E',
$w(u,v) + d_T(s,u) \geq d_T(s,v)$ … 1
(w(u,v) is the weight of the edge and $d_T(s,u)$ us the distance between nodes s u along the edges of T and similarly for $d_T(s,v)$. We can calculate d_T using BFS)
If 1 is false for at least one edge that does not belong to T, then T is not the shortest path tree.

Algorithm:

```
shortest_path(G, T):
Run BFS(T) from s to calculate $d_T(s,x)$ which is the distance from node s to node x along the edges of T.
for edge from edges not in T:
    if $w(u,v) + d_T(s,u) < d_T(s,v)$:
        return false
return true
```

Correctness:
If T is a shortest path tree, it must hold that $w(u,v) + d_T(s,u) \geq d_T(s,v)$ between nodes s and u along the edges of T. If this is not true, the algorithm returns false as T will not be the shortest path tree.
If it is the shortest path tree, the above formula will hold and the algorithm will return true if it evaluates all the edges that are not part of tree.

Running Time: The algorithm runs for each edge for each vertex in the graph. This means that it will run in linear time in $O(|V|+|E|)$.

23)
We have a directed graph G = (V,E). We want to take this graph and return the length of the shortest cycle in the graph if it exists. If it is acyclic, we can return 0 or false. A shortest cycle a path from vertex 1 to vertex 2, followed by an edge between vertex 2 and 1.

Algorithm shortest_cycle(G,s):

25)
To get the right output for each vertex, we can make a simple modification to the dijkstras algorithm to solve this.

```
Algorithm modified_dijkstra(G, le, c, s):
for v in V:
dist(u) = MAXINT
previous(u) = null
dist(source) = cost(source)
Let L be the list of all nodes in the graph.
while Q !empty:
        u is the node with the small distance in dist[]
        remove u from Q
        if dist[u] == infinity:
                return dist
        for neighbor in neighbors(u):
                alt = dist(u) + len(u,v) + c(v)
                if alt < dist(v):
                        dist(v) = alt
                        previous(v) = u
                        decrease-key v in Q
return dist
```

Correctness:

The algorithm runs a modified form of dijkstras. It runs a loop to check that all vertices are being processed and it returns the array which has the least cost from s to any vertex. Since it goes through the vertices systematically, it checks if the distance is lower than the preset distance. If it is, it will be replaced in the main array.

Running Time:
The total vertices are |V|, the total edges are |E|.
The heap operation takes log|V| time. With this we get a total running time of
O((|V|+|E|)log|V|)

26)
a) Since the cycle is of negative weight, r* is the maximum ratio achievable from any simple cycle and r is less than this ratio. Therefore r < r* if there is any negative weight cycle.

b) When every cycle has positive weight, then r will be greater than the above fraction of profit by cost for every cycle. Since r* is maximum value obtained by a simple cycle. r > r*.

c) Given R which is the max ratio of profit and cost of some edge, this will be the upper bound for r*. Now make edge weight of every edge equal to profit by cost ratio. We an find the cycle of max weight.
Algorithm:
For each vertex v in graph, find the longest weight cycle. This can be done by finding the longest path to vertex u, where u has edge towards v and then add the weight from u to v.
For the cycle we get from step 1, we will test if profit by cost ratio is ≥ r*-E (symbol given in pdf). If it is satisfied, then we return this cycle. Else no cycle can satisfy this ratio ≥ r*-E.

Correctness:
This algorithm will function properly as we are considering cases in which the ratio is greater than, equal to and less than

Running time:
For each vertex v, we can find the max weight cycle in O(n^2) time, where n is the number of vertices and exploring all paths take O(n) time. Doing this for each vertex will take a total of O(n^3) time.

27)
a) Given any clique in the graph, we can easily verify in polynomial time that there is an edge between every pair of vertices. Therefore a solution to CLIQUE-3 can be checked in polynomial time. This proves that clique is in NP.

b) The problem is being reduced incorrectly. CLIQUE must be reduced to CLIQUE-3 instead of CLIQUE-3 to CLIQUE to show that CLIQUE-3 is at least as hard as CLIQUE.

c) The statement “a subset C ⊆ V is a vertex cover in G if and only if the complementary set V-C is a clique in G” is false. C is a vertex cover if and only if V-C is an independent set in G.

d) If a vertex has a degree of at most 3, there is no way for there to be a clique of size 5 greater. Checking if the vertex is a part of a clique size 4 or smaller is a O(|V|) operation.

28)
We know that 3SAT is NP-Complete. To show that 4SAT is NP-complete, we will show that it is in NP and NP-hard.
To show that 4SAT is in NP, we can write a nondeterministic polynomial time algorithm which takes a 4SAT instance and a proposed truth assignment as input. This algorithm will evaluate this instance with the truth assignment. If the instance evaluates to true, the algorithm gives us true, otherwise it outputs no. Such an algorithm will run in polynomial time, proving that 4SAT is in NP.
To prove that 4SAT is NP-Hard, we can reduce 3SAT to 4SAT. Let m denote an instance of 3SAT. We convert m to a 4SAT instance m' by turning each clause (x and y and z) in m to (x or y or z or h) AND (x or y or z or not h) where h is a new variable. We can do this in polynomial time.

If a given clause (x or y or z) is satisfied by a truth assignment, then (x or y or z or h) AND (x or y or z or not h) is satisfied by the same truth assignment with h arbitrarily set. Thus if m is satisfiable, m ' is satisfiable.
Suppose m ' is satisfied by a truth assignment T. Then (x or y or z or h) AND (x or y or z or not h) must be true under T. As h and ¬h assume different truth values, x or y or z must be true under T as well. Thus m is satisfiable.
As this is proved, 4SAT is NP-hard.
As 4SAT is both in NP and NP-hard, it is NP-complete.

29)
For each edge in E, we can choose nondeterministically if the edge is to be included in T. Then we can check if T is a tree and confirm that each vertex has degree less than 2. This can be done in polynomial time. This means that the 2-SPANNING TREE problem is in NP.
We have to find a tree with each vertex having degree at most 2. However, such a tree must be a path, since creating a branch at any vertex makes that vertex of degree 3. Also, if the tree spans the graph, it must be an undirected Hamiltonian Path. We already know that the Hamiltonian path problem is NP-Complete. We have reduced HAMILTONIAN PATH to the 2-SPANNING TREE problem. Therefore the 2-SPANNING TREE problem is NP-complete.

30)
Considering two graphs G1 = (V1,E1) and G2 = (V2,E2), we can assume that here is a mapping M from vertices of the subgraph G1 to subgraph G2. The mapping M can be checked as an isomorphism by an algorithm in polynomial time. This proves that MAXIMUM COMMON SUBGRAPH is in NP.

We can reduce SUBGRAPH ISOMORPHISM TO MCG in polynomial time to show that MCG is NP-hard.
Let G1 and G2 be the inputs to SUBGRAPH ISOMORPHISM.
Let G1,G2, |V1| be the input to MCG.
There is a subgraph of G2 which is isomorphic to G1 iff there are subsets W1 ⊆ V1, and W2 ⊆ V2 who will leave behind at least |V1| vertices in each graph when deleted. This will make the two graphs

identical to each other. Also |W2| = |V2|-|V1|. This shows that reduction in polynomial time is possible. Therefore, we know that MCG is NP-hard.
As MCG is both in NP and NP-hard, we prove that MCG is NP-Complete.

32)
a) Given a graph and a FAS, we can check that the size of the set is at most b and the graph produced by removing the edges in the set is acyclic using DFS. This can be done in polynomial time. So FAS is in NP.

b)We can say that if U ⊆ V is a vertex cover for G, then F = {(wi,wi')|vi is in U} is a FAS of the same size for G'.
To prove this we note that G' is a bipartite graph with w1,…,wn on the left and w1',…,wn' on the right. A cycle must involve an edge e.g. (wi',wi) from right to left. However, the only incoming edge into wi' is (wi,wi') and the only outgoing edge from wj is (wj, wj'). This means both of them must be in the cycle. Also since (wi',wj) is an edge, (vi,vj) in E and either vi or vj must be in U. which means F intersection {(wi,wi'),(wi,wi')} != 0. Therefore, removing the edges in F breaks this cycle.

c) We can replace all the edges of the form (wi',wi) in the given FAS, by edges of the form (wk',wk). By the same argument as part b, any cycle containing (wi',wj) must also contain (wi,wi') and hence F\{(wi',wj)} union {(wi,wi')} is also a FAS. In this way, we never increase the size of the set, but might decrease it. We can claim that U = {vi| (wi,wi') in F'} must be a vertex cover for G. If it is not, then there has to be an edge (vj,vk) such that vj,vk not in U and hence (wj,wj'), (wk,wk') not in F'. But then wj -> wj' -> wk -> wk' -> wj would be a cycle in G' even after removing F', which is a contradiction. Thus proved.

33)
Let T be the minimum steiner tree with cost C. We can follow the shape of a steiner tree to obtain a tour of cost 2C which passes through all vertices in the steiner tree. Let i,j,k be adjacent vertices in the path. Using the triangle inequality, we know that dik $\leq$ dij + djk. Using this, we can bypass a middle vertex j and connect i and j directly, without increasing the cost. Using this method, we can bypass all vertices V' that are present in the path and also the vertices of V' which are visited twice. This gives a path of cost at most 2C, which passes through all the vertices of V' exactly once. So we get a spanning tree for V' of cost 2C. This implies that cost MST $\leq$ 2C. Giving us an efficient ratio-2 approximation algorithm.

34)
a) If k=2, multiway cut become the same as the min-cut problem in a max flow graph. A max flow graph can be solved in polynomial time. Hence, it is proved that multiway cut can be solved in polynomial time if k = 2.

b) Algorithm:
```
multiway_cut(G, s1, s2, s3):
For s1,s2,s3
create vertex t
connect t to both nodes except the loop node.
set weight of edge from t to both non loop nodes = inf
Find min-cut(loop node,t) of the new graph to separate loop node from all other terminal vertices.
Let E(loop node) be the set of edges in min-cut(loop node,t) and d(E(loop node)) is the value of the cut.
```

With this we get E1, E2, E3 as output.

Correctness:
Suppose that opt-cut is the best 3-way cut to separate s1 and s2 terminals.
So, opt-cut ≥ d(Ei), for i = 1,2,3
In a 3-way cut, at least 1 terminal needs to be separated from the others.
To separate E1 from E2 and E3, assume that E2 union E3 is U.
d(U) ≤ d(E2) + d(E3)
We know that
d(E2) + d(E3) ≤ 2(opt-cut)

from these equations, we get
d(U) ≤ d(E2 + E3) ≤ 2(opt-cut)
This shows that the algorithm has a ratio of at most 2.
Proved.

35)
a) We introduce 0-1 valued variables yj that indicate whether investor j is used or not and 0-1 valued variables xi, that indicate whether actor i is used or not.
Integer linear program:
max $m\sum_{j=1}$ykPj - $m\sum_{i=1}$xisi

for all j,i in Lj : yj ≤ xi
for all i,j: xi, yj in {0,1}

The constraints ensure that yj will be 1 only if all actors required by producer j are included.

b) We will not relax this into a linear program by changing by hanging the last constraint to:
for all i,j : 0 ≤ xi,yi ≤ 1. We will show that there is an integral optimal solution.
Consider an optimal solution to the LP which is not integral and take the highest xi in (0,1).
There may be multiple such xi tied at the same highest value, v and lets call the set of such i's, A.
Now for each i in A, there must be j such that i in Lj and yj = v. Otherwise we can decrease xi to get a higher value of the objective function. Let B be the set of all such j's. Therefore these sets yield a contribution to the objective function proportional to w = $\sum_{j \text{ in } B}$Pj - $\sum_{i \text{ in } A}$si
If w is negative, we derive a contradiction by lowering the values corresponding to all members of A and B by an equal infinitesimal amount, which will not violate constraints. If w is positive, we raise these values to raise the objective function. If w is zero, we have a non-unique solution. But we can take all values of members in A and B to be 1 to achieve the same objective. If we repeat for any other xi in (0,1) will give us an integral solution.