COL351 Assignment2

Aayush Goyal 2019CS10452
Aniket Gupta 2019CS10327

September 2021

1 Algorithms Design book

We will denote $\Sigma_{i\in S_1}x_i, \Sigma_{i\in S_2}x_i$ and $\Sigma_{i\in S_3}x_i$ by sum_1, sum_2 and sum_3 respectively. Also, the sum of no. of questions in first i chapters is denoted by totalQuestions(1:i).
Now, the basic idea is to check if it is possible to partition the chapters in three parts for all the possible values of sum_1, sum_2 and sum_3 and then chose the best among them.
In our solution, **partition(n, sum_1, sum_2, sum_3)** returns True only if there is a partition possible for the first n chapters with the total no. of questions in the three sets equal to sum_1, sum_2 and sum_3 respectively. Otherwise, it returns False
We use the following observations to obtain a recursive solution of partition(n, sum_1, sum_2, sum_3).

- **Observation 1:** If n=0, then there are zero chapters and a partition is possible only when $sum_1 = sum_2 = sum_3 = 0$ (since all the three sets will be empty).
- **Observation 2:** If any of the sum_1, sum_2 and sum_3 is negative, then partition is not possible because no. of questions in a chapter is always non-negative.
- **Observation 3:** If $sum_1+sum_2+sum_3$ is not equal to the total no. of problems in the first n chapters, then such partition is also not possible. This is because every chapter must be in one of the three sets and thus sum of total no. of problems in each set is equal to the total number of problems in the n chapters.
- **Observation 4:** Let us assume that there is a partition possible for the first n chapters with the total no. of questions in the three sets equal to sum_1, sum_2 and sum_3 respectively. Then, the last chapter will be in any one of the three sets. Without loss of generality, let us assume that last chapter is in set S_1. Then the sum of problems of the chapters in S_1 (except last chapter) will be sum_1-x_n. Thus, there would be a way to partition the first n-1 chapters in three sets such that total no. of problems in the three sets are sum_1-x_n, sum_2 and sum_3 respectively. Since the last chapter could have been in any of the three sets, we can write **partition(n, sum_1, sum_2, sum_3) = partition(n-1, sum_1-x_n, sum_2, sum_3) or partition(n-1, sum_1, sum_2-x_n, sum_3) or partition(n-1, sum_1, sum_2, sum_3-x_n).**

Based on the above observations, we can write the recursive relation to check if such a partition exists for a given value of n, sum_1, sum_2 and sum_3 as:

partition(n, sum_1, sum_2, sum_3) = **True** if (n = sum_1 = sum_2 = sum_3 = 0)
= **False** if ((n = 0) and ($\text{sum}_1 \neq 0$ or $\text{sum}_2 \neq 0$ or $\text{sum}_3 \neq 0$))
or $\text{sum}_1 < 0$ or $\text{sum}_2 < 0$ or $\text{sum}_3 < 0$
or sum_1+sum_2+$\text{sum}_3 \neq$ totalQuestions(1:n)
= **partition(n-1, sum_1-x_n, sum_2, sum_3)**
or **partition(n-1, sum_1, sum_2-x_n, sum_3)** otherwise
or **partition(n-1, sum_1, sum_2, sum_3-x_n)**

Note that the **or** and **and** in the above equation are logical operators.

- **Observation 5:** In Observation 3, we mentioned that if sum_1+sum_2+sum_3 is not equal to the total no. of problems in the first n chapters, then there is no partition possible. Thus, for valid partitions of first n chapters, sum_1 and sum_2 uniquely determines sum_3 (= totalProblems(1:n)-sum_1-sum_2). Therefore the above recursive relation can be modified to obtain the below recursive relation with one less variable.

partition(n, sum$_1$, sum$_2$) = **True** if (n = sum$_1$ = sum$_2$ = 0)
= **False** if ((n = 0) and (sum$_1$ $\neq$ 0 or sum$_2$ $\neq$ 0)
or sum$_1$ < 0 or sum$_2$ < 0
or sum$_1$+sum$_2$ > totalQuestions(1:n)
= **partition(n-1, sum$_1$-x$_n$, sum$_2$)**
or **partition(n-1, sum$_1$, sum$_2$-x$_n$)** otherwise
or **partition(n-1, sum$_1$, sum$_2$)**

Note that for the cases when partition is possible, both the recursive relations are equivalent. This is because whenever partition(n, sum$_1$, sum$_2$, sum$_3$) is true according to first equation, then sum_3 (= totalProblems(1:n)-sum_1-sum_2) is uniquely determined.

Once, all the valid values of sum$_1$ and sum$_2$,that gives a possible partition, are determined, we check for the partition with minimum value of maxsum$_1$, sum$_2$, sum$_3$ (Note: sum$_3$=totalQuestions(n)-sum$_1$-sum$_2$) and return that partition. Also, note that for each valid value of sum$_1$ and sum$_2$, we also need to store the set in which the current chapter is included to return the final partition.
This recursive relation can be implemented using dynamic programming as shown below:
Algorithm:

Proof for correctness of the algorithm:
In our algorithm, partition(n, sum$_1$, sum$_2$) used in the recursive equation is stored in partition[n][sum$_1$][sum$_2$].

- We first show that algorithm correctly determines partition[i][j][k] (=partition(i, j, k)) for 0<=i<=n, 0<=j<=totalProblems and 0<=k<=totalProblems and setNo[i][j][k] correctly stores the set in which chapter i will be taken if partition[i][j][k] is true. We use principle of induction on the value of i to prove this.
  Base Case: partition[][][] is initialised to false in line 2 and partition[0][0][0] is marked True in line 7. Also, setNo[0][i][j] is -1 for all i and j in given range. This is correct since there is no chapter zero and thus cannot be placed in any of the three sets. Thus, for case when n=0, the algorithm is correct.
  Induction Step:
  Let us assume that the algorithm correctly computes partition[i][j][k] and setNo[i][j][k] for some 0<=i<n. We will show that the algorithm correctly computes partition[i+1][j][k] and setNo[i+1][j][k] as well (here j, k can be any value in the given range).
  Note that the chapter no. goes from 1 to n in the outer loop in line 24. Thus, before the (i+1)th iteration of the outer loop (line 24), partition[i][j][k] and setNo[i][j][k] are already calculated for any value of j and k in the given range and by the inductive assumption, these values will be correct.
  In lines 27-28, we do not change anything if sum of problems in first two sets is more than totalQuestion(1:i+1). According to the recursive relation, in this case, partition is not possible and initially the partition[i+1][j][k] was initialised to false. Since, we are not changing anything in this case, the algorithm is correct for this case.
  Lines 29-33 checks the condition if it is possible to partition the first i+1 chapters (for given values of j and k) such that the (i+1)th chapter is taken in set 1. By recursive relation, we know it is possible only if partition(i, j-x$_{i+1}$, k) is True. By our inductive assumption, partition[i][j][k] was correctly calculated for all values of j and k in the given range and thus partition[i][j-x$_{i+1}$][k] was correctly determined. Since, we already know partition(i, j-x$_{i+1}$, k), we correctly compute partition[i+1][j][k] and modify setNo[i+1][j][k] to 1. Similarly, lines 34-42 checks if a partition is possible with chapter i+1 in sets 2

and 3. Note, since it is true for any arbitrary j and k in the given range, it is true for all the values of j and k in the given range.
Thus, the algorithm correctly computes partition[i+1][j][k] and setNo[i+1][j][k] for all values of j and k.
Thus, by principle of induction, the claim is true.

Now, using the fact that partition[i][j][k] is correct for all i, j, k in the domain. We can iterate over all the values of j and k for i=n to get the partition which has minimum value of (i, j, totalProblems-k) (in line 51-56).
Once, we know the optimal value of sum_1, sum_2 and sum_3, we can backtrack to get the chapters in each set. In lines 58-69, we start from chapter n and move downto chapter 1. In the first iteration, we correctly get the set in which chapter n can be taken since setNo[n][sum_1][sum_2] correctly store the set no. in which chapter n can be taken as proved above. Now, the values of sum_1 and sum_2 is modified for which there is possible partition according to the recursive relation. In the next step, set for chapter n-1 is determined. Repeating this step iteratively gives the correct set for all the chapters.
Thus, the algorithm correctly determines the partition which minimises the max(sum_1, sum_2, sum_3).

Time complexity:
Note that no. of problems in a chapter is bounded by n (no. of chapters). Thus total no. of problems in the book is $O(n^2)$. Thus, the loop in lines 24-42 takes $O(n*n^2*n^2) = O(n^5)$ time. Now, finding the optimal values in lines 51-56 takes $O(n^2*n^2) = O(n^4)$ time. The backtracking to get the partition in lines 58-69 takes O(n) time. Thus, the total time complexity = $O(n^5) + O(n^4) + O(n) = O(n^5)$ time.

2 Course Planner

Problem Formulation:
This problem can be reformulated in a directed graph problem. Construct a directed graph with vertex set C and edge set E such that (x,y)∈E (directed edge from x to y) iff x is a prerequisite of y, i.e. x∈P(y).
Claim 1:
If there is any path $c_1 -> c_2 -> c_3 ... -> c_{t-1} -> c_t$ in the above defined directed graph, then the courses must be done in the order $c_1, c_2, c_3 ..., c_{t-1}, c_t$.
Proof for claim 1:
According to the definition of the edge in the above graph, c_i is a prerequisite of c_{i+1} in this path ∀ i∈(1,2,3,...t-1) and thus course c_{i+1} can be taken only if c_i is already completed. Thus, the way in which the edges are defined imply that the courses must be done in the order $c_1, c_2, c_3 ..., c_{t-1}, c_t$.

2.1

Claim 2: An order for taking the courses, so that a student is able to take all the n courses with the prerequisite criteria satisfied, exists if and only if there is no cycle in the above defined directed graph.
Proof for claim 2:

Proof for forward implication:
We prove "If there is required ordering available, then there is no cycle in the graph defined" by using contradiction.
Let us assume such an ordering exists but there is a cycle in the directed graph. Let that cycle be $c_1 -> c_2 -> c_3 ... c_n -> c_1$. Then by claim 1, the order of completion must follow $c_1, c_2, c_3 ... c_k, c_1$. This implies that c_1 must be completed before c_1 which is not possible. Therefore, there is no such ordering and it contradicts our assumption. Hence, there cannot be a cycle in the directed graph defined if there is a required ordering available.

Proof for backward implication:
We prove "If there is no cycle in the graph defined, then there is a required ordering available".
Run a dfs traversal on the directed graph defined. Since the graph is acyclic, therefore there will be no back-edges in the dfs traversal tree of the graph. Now, arranging the courses in decreasing order of finish time, we get an ordering (topological ordering) such that for all edges (u− >v) in the graph, u occurs before v in the ordering. Now, we will prove inductively that taking the courses in the topological ordering obtained, a student can complete all the courses with prerequisites completed.
Let the topological ordering be v1, v2, v3, ... vn
Induction predicate:
H(i) := If we take the courses in topological ordering, then at the time when we take course v_i, all its prerequisites will be already satisfied.
Base Case:
Since, in a topological ordering, all the edges go from left to right therefore there is no node which has an from itself to v1. Thus, there is no prerequisite for course v1 and it can be taken in the beginning.
Induction Step:
Let us assume that H(j) is true for all 0<j<=i for some i (0<i<n). Then, we will prove that H(i+1) is also true.
Since in a topological ordering, all the edges are from left to right therefore, all the nodes which have an edge from itself to v_{i+1} will lie before v_{i+1} in the topological ordering. Thus, all the prerequisites of v_{i+1} will lie before v_{i+1} in the topological ordering. Therefore attending the nodes in topological ordering, we get that prerequisites of v_{i+1} will be already visited before v_{i+1}. Also from our inductive assumption, conditions for all the prerequisites of v_{i+1} were already satisfied when they were visited in the topological order and thus H(i+1) is also true.
Thus, we proved that "If there is no cycle in the graph defined, then there is a required ordering available".

This completes the proof for claim 2.

Algorithm:
The basic idea is to check if the graph obtained is a directed acyclic graph and if it is so, find a topological ordering for the graph.

Time Complexity:
Creating the adjacency list (outAdj) and edge set from the given problem will take O(m) time where m is the no of (prerequisite, course) pairs in the original problem. If the no. of courses are n, then dfs traversal will take O(n+m) time. Checking for back edges takes O(m) time. Then, sorting the nodes according to decreasing order of endTime can be done using bucket sort in O(n) time. Thus, the overall time complexity of the algorithm is O(n+m).

2.2

We are assuming that a valid ordering exists which allows to complete all the courses with the prerequisites complete. If there is no such ordering, it can be detected using algorithm in part 2.1
Claim 3: For any course c, the minimum no of semesters required to complete that course is one more than the length of longest path ending at c in the directed graph obtained.
Proof for claim 3:
Since we are assuming that an ordering exist, then the graph is acyclic and a topological ordering exist (by claim 2). Let the topological ordering obtained be v1, v2, v3, ... vn. We prove claim 3 by induction on the topological ordering.
Induction Predicate: H(i) := minimum no. of semesters required to complete course v_i is one more than length of longest path ending at v_i.
Base Case: Since in a topological ordering, all the edges go from left to right,the longest path ending at v1 has length 0. Now, since there is no. node which has an edge from itself to v1, v1 has no pre-requisite and thus it can be completed in the first semester itself which is one greater than length of longest path ending at v1. Thus, H(1) is true.
Induction Step:
Let us assume that H(j) is true for all 0<j<=i for some i (0<i<n). We will prove that H(i+1) is also true. Now the minimum no. of semesters required to complete the course v_{i+1} is obtained if v_{i+1} is taken in the very next semester when all it prerequisites are completed. Let the prerequisites of v_{i+1} be v_{k_1}, v_{k_2}, v_{k_3} ..., v_{k_t}. According to the way the graph edges are defined, there is an edge from each of these prerequisites to v_{i+1}. Thus, in the topological ordering, all these nodes appear before v_{i+1}. Among all the prerequisites of v_{i+1}, say v_{k_l} has the longest path ending at that node, say of length l'. Then, v_{k_l} will take l'+1 semesters for its completion according to induction hypothesis. Also, minimum no. of semesters for the other prerequisites will be less than or equal to l'+1 due to the induction hypothesis and the assumption that among the prerequisites of v_{i+1}, v_{k_l} has the longest path ending at that node. Now, the minimum no. of semesters to complete v_{i+1} will be l'+2. Also, all the paths that end at v_{i+1} will have one of the prerequisites as the second last node in the path (if path length is greater than 1). Thus the length of longest path ending at v_{i+1} is one more than the length of longest path ending at any of its neighbours. Therefore, length of longest path ending at v_{i+1} is l'+1. Thus minimum no. of semesters required to complete v_{i+1} is one more than the length of longest path ending at v_{i+1} [l'+2 = (l'+1)+1]. Thus, H(i+1) is also true.
Thus, by principle of induction, we proved the given claim.

Now, the minimum no. of semesters required to complete all the courses is one more than length of the longest path ending at any of the course. Thus, we need the length of longest path in the directed graph obtained.
Algorithm:

Correctness of algorithm:
For this part, I am assuming that the graph has no cycle. If there is a cycle in the graph, then it can be detected using algorithm in part 2.1. First we obtain a topological ordering of the courses such that for any edge (u− >v), u occurs before v in the ordering. We prove that this algorithm gives the minimum no. of semesters required to complete all the courses.
To prove this, we first show that the loop invariant (for loop in line 11-14)- "After i^{th} iteration of outer loop, length of longest path ending at first i courses in the topological ordering is correctly calculated in longestPath[] array".
We prove this by induction on the above loop invariant mentioned.
Let the topological ordering obtained be v1, v2, v3, ... vn.
Base Case:
Before the first iteration, longestPath[v1] = 0. Also, since for any edge (u− >v), u occurs before v in the ordering, there is no node which has an edge from itself to v1. Thus, the inner loop (line 12-14) is executed for zero times and therefore after the first iteration, longestPath[v1] is still zero, which is precisely the length

of longest path ending at v1.
Induction step:
Let us assume that longestPath[j] is correctly computed for all $0 < j <= i$ for some i (0<i<Sn) before the (i+1)th iteration of the outer loop. Then, during the (i+1)th iteration, all the prerequisite courses are visited in the inner loop. Since the nodes are being traversed in topological order, all its prerequisite courses of v_{i+1} will appear before v_{i+1} in the ordering and thus the longest path for these prerequisite course nodes is correctly calculated due to the induction assumption. Now, length of longest path ending at v_{i+1} is one more than length of longest path ending at any of its prerequisite. Thus, the line 12-14 correctly computes the length of longest path ending at v_{i+1}. Thus, after the (i+1)th iteration of the outer loop, length of longest path ending at v_{i+1} is correctly calculated.
Thus, be principle of induction, the loop invariant holds after each iteration and thus at the end length of longest path ending at all the courses is correctly obtained.

Now, the line 16-18 finds the length of longest path ending at any node in the graph. Then line 20 increases the sol by 1 since minimum no. of semesters required is one more than the length of the longest path in the graph.
Thus, the given algorithm is correct.

Time complexity:
Finding the topological ordering takes O(n+m) time. Now, the inner for loop in lines 12-14 visits prerequisites courses of the current course taken in the outer for loop. Thus, the total time taken for line 11-14 is O(n+ m) in the graph (n is total no. of courses and m is the total no. of edges in the graph obtained). Also, the loop in line 17-18 takes O(n) time. Thus, total time complexity is O(n+m).

2.3

Claim 4: For two courses c and c', L(c)∩L(c') is empty if and only if there is no node from which there is a path to both c and c' in the directed graph obtained.

Proof for claim 4:

If there is a path from any node v' to another node v", then v' must be completed before v" (see claim 1). Thus, v'∈L(v").

If L(c)∩L(c') is empty, then there is no node, say v, from which there is a path to both c and c'.

We prove this by contradiction. Let us assume that L(c)∩L(c') is empty but there exist a node v from which there is path to both c and c'. Then, according to claim 1, v must be completed before c and v must be completed before c' as well. Thus, by definition of L(c), v∈L(c) and v∈L(c') which implies that v∈L(c)∩L(c') which contradicts the fact that L(c)∩L(c') is empty. Thus there is no such node v from which there is a path to both c and c'.

If there is no node from which there is path to both c and c' then L(c)∩L(c') is empty.

We prove this by contradiction. Let us assume that there is no such node from which there is path to both c and c' but there is a course v such that v∈L(c) and v∈L(c'). If v∈L(c), then either v is one of the prerequisites of c or v is required to complete one of the prerequisites, say c_1, of c. By similar argument, we can say if v is required to complete c_1, then either v is one of the prerequisites of c_1 or v is required to complete one of the prerequisites, say c_2, of c_1. If we keep repeating this step, we will get a sequence c_k = v, c_{k-1},..., c_3, c_2, c_1 (=c) such that c_t is a prerequisite of c_{t-1} for all 1<t<=k. Now by definition of the graph edge in problem formulation, there is an edge from c_t to c_{t-1} for all 1<t<=k and thus there is a path from v to c in the graph. Similarly, since v∈L(c'), there is a path from v to c' in the graph. But this contradicts the assumption that there is no node from which there is path to both c and c'. Thus, our assumption cannot be true. Hence, if there is no node from which there is path to both c and c' then L(c)∩L(c') is empty.

This completes the proof of claim 4.

Approach for the algorithm:

For every node, say v, we find the nodes that are reachable from v in the directed graph. This can be done in O(n+m) time for a single node. Computing this for all the nodes will take O(n*(n+m)) time. Then for every pair (x,y), we check if L(x)∩L(y) is empty by checking if there is some k from which there is a path to both x and y. There are O(n^2) pairs and for each pair we have to check for n values of k.

Algorithm:

Proof of correctness:

The dfs traversal in lines 8-12 visits all the nodes that are reachable from current node. In lines 15-20, after the nodes reachable from current node, say v, are found by the dfs traversal, we mark reachable[v][w] = 1 if there is a path from v to w in the graph. Finally, after the loop in lines 14-20 is completed for all the nodes, we have list of nodes reachable from each node (in the form a two dimensional matrix reachable[][]) for all c∈C.

Now, in lines 25-33, we consider all the possible pairs of nodes (i,j). In lines 28-29, we check is there is any node k from which there are paths to both i and j. If there is such a node, we break the inner loop and do not include it in the solution otherwise we include (i,j) in the solution set (this is true by claim 4). This proves that the algorithm correctly finds all such pairs.

Time complexity:

There are n dfs traversals in lines 14-20 to find the list of nodes reachable from each node. Time for one dfs traversal is O(n+m). Total time for n dfs traversals is thus O(n*(n+m)). Now, we consider all the pairs in lines 25-33 and for each pair (i,j) we check if there is a node k from which there is a path to both i and j. There are O(n^2) pairs and n values of k for each pair. Thus, lines 25-33 takes O($n^2 * n$) time. Thus, total time complexity is O($n * (n + m) + n^3$) = O(n^3) time (since m = O(n^2)).

3 Forex Trading

The problem can be visualized as a directed graph, where if there is an edge from node 'i' to node 'j' then, it's cost is represented by R(i,j). For the problems below we will be converting the weight of edges into another form. Every edge weight will changed to R*(i,j), where **R*(i,j)** $= -\log(R(i,j))$

3.1

We need to find a cycle whose edges satisfy the property that the product of reward of all it's edges is greater than 1. Thus if the cycle contains the vertices $i_1, i_2, ..., i_k, i_{k+1} = i_1$, then we nedd a cycle such that the rewards satisfy $R[i_1, i_2] * R[i_2, i_3] * \cdots * R[i_k, i_1] > 1$. In this relation, take $-\log$ on both the sides. Since log is a increasing function, we have the following $-\log(R[i_1, i_2] * R[i_2, i_3] * \cdots * R[i_k, i_1]) < -log(1)$. This can be written as $(-\log(R[i_1, i_2])) + (-\log(R[i_2, i_3])) + \cdots + (-\log(R[i_k, i_1])) < 0$. Each of the term lhs can be written as R*[i,j] (the changed reward we defined above). Now the problem comes becomes to find if there is any cycle with total weight negative in the graph with new wegiht edges, given by R*[i,j] $= -\log(R[i,j])$. This becomes a standard problem now to determine if there is any negative weight cycle in the graph or not. This can be done by modifying bellman ford-algorithm a little bit.

Claim: G contians a 'negative weight cycle' if and only if we are able to make an imrpovment even in the Distance vector even in the n^{th} round.
Proof of Claim: The shortest path from the starting vertex to some vertex 'v' can have atmost 'n-1' edges. This is because, the maximum number of edges we can have on a simple path is (n-1). A tree contains unique path from root to any vertex and thus we cannot require more than n-1 edges on whatever is the shortest path. The only way this can happen is that we were struck in a negative weight cycle and we kept iterating over the negative cycle to continuously keep decreasing the cost of path. Hence if there is no negative cycle in the graph, their should not be any improvement in the distance vector after n-1 iterations. But if we are able to make any improvement in the n^{th} iteration, that means the shortest path from root to vertex is of length more than $n-1$ edges and this can only happen when the path from root to vertex is not simple. We have encountered a negative cycle in the path and are walking through it to get further reduced distance.

Algorithm Outline:
We will use the bellman ford algorth to determine of there is any negative weight cycle in the graph or not. The basic idea is, after running the $n-1$ ($n = |V|$) iterations of bellman ford algorithm, if the graph contains a negative weight cycle then by Claim 1, we are able to make an improvement even in the n^{th} iteration. The graph here is assumed to be connectedd since it represents the exchange rates between currencies.

Algorithm:

```
edges <- contains the edge list
R <- the weight of edges, R[i,j] denotes the weight of edge from  i to j
R* <- this will store the modified edge weights

distance <- this contains the distance of each node from the starting vertex
            (Every distance is initialized by infinity)

N <- |V|    # number of edges in the graph


# Making the R* vector
for i<-0 to N-1 do:
    for j<-0 to N-1 do:
        R*[i,j] = -1*log(R[i,j])
```

```
15
16
17 # initialing the distance vector
18 for i<-0 to N-1 do:
19     distance[i] <- inf
20
21 start <- start is the starting node
22
23 # Now doing the first N-1 iterations of bellman ford algorithm
24 for i<- 1 to n-1 do:
25     for edge in edges do:
26         u  = edge.first  # the starting vertex
27         v  = edge.second # the ending vertex, means edge from u to v
28         w = R*[u,v]
29         if (distance[v] >  distance[u] + w) then:
30             distance[v] = distance[u] + w
31
32 # Now doing the Nth iteration to find if we are still able to make any
33 # improvement, if we are able to make an improvement that means the graph
34 # has a negative weight cycle.
35
36 hasNegCycle <- False
37
38 for edge in edges do:
39     u  = edge.first  # the starting vertex
40     v  = edge.second # the ending vertex, means edge from u to v
41     w = R*[u,v]
42     if (distance[v] >  distance[u] + w) then:
43         hasNegCycle <- True
44         break
45
46 # if hasNegCycle is True that means we found a negative weight cycle in the
47 # graph. This means there exists a cycle which has product of the R[i,j] of
48 # edges greater than 1.
```

Time complexity analysis:
The time complexity of the following algorithm is $O(mn)$, n is the number of vertices, m is the number of edges which here is $O(n^2)$. The breakout of complexity is as follows:

1. Making R* from takes $O(n^2)$ time since we iterate with 2 nested for loops.
2. Now the main part of algortihm from line 24-30, we run the outer loop $n-1$ times and in each for loop, we iterate over the entire edge set and thus in the inner loop we do m iterations. Thus overall we do $(n-1)*m$ iterations.
3. In line 38-44 we do m iterations, since we just iterate over the entire list once
4. Now combining the point 2 and 3 above we have $n*m$ iterations, and thus it takes $O(nm)$ time.

Thus the overall time complexity of the process becomes $O(mn+n^2)$ and if we consider this as a compelte graph then $m = O(n^2)$ and hence the overall time complexity is $O(n^3)$

3.2

In the above we used Bellman ford algorithm to determine if there exists a cycle with negative edge weight. In this part we will print the cycle if it exists and -1 if it doesn't exist.

Claim 1: If a vertex v encounters improvement in the n^{th} iteration of Bellman-Ford, then there is a path P from source vertex s to v such that P contains a negative weight cycle.
Proof for claim 1:
We prove this claim by using contradiction. Let us assume that vertex v encounters improvement in the n^{th} iteration of Bellman-Ford, but there is no path P from source vertex s to v that contains a negative weight cycle. Now, in Bellman-Ford algorithm, any vertex for which a shortest path from source s is a simple path with k edges is detected in maximum of k iterations. Since vertex v encounters improvement in n^{th} iteration, the shortest path from s to v has atleast n edges. But in a graph with n vertex, any simple path will have atmost n-1 edge. Thus, the shortest path from s to v cannot be a simple path. Also, shortest path between any two vertex in a directed graph never contain a cycle of positive weight. This

Claim 2: If a vertex v encounters improvement in the n^{th} iteration of Bellman-Ford, then back-tracking for n times from vertex v, along the parent nodes, in the predecessor graph of Bellman-Ford, we reach a vertex that is a part of a negative cycle.
Proof for claim 2:

Algorithm outline:
We will need to store the parent of each vertex while we are running the iterations of Bellman Ford algorithm (i.e. during the first n-1 iterations). Now when we are running the n^{th} iteration we will need to store the first vertex where we encountered an improvement.

Algorithm:

```
edges <- contains the edge list
R <- the weight of edges, R[i,j] denotes the weight of edge from  i to j
R* <- this will store the modified edge weights

distance <- this contains the distance of each node from the starting vertex
            (Every distance is initialized by infinity)

parent <- this is to store the parent of each vertex

N <- |V|    # number of edges in the graph


# Making the R* vector
for i<-0 to N-1 do:
    for j<-0 to N-1 do:
        R*[i,j] = -1*log(R[i,j])


# initialing the distance and parent vector
for i<-0 to N-1 do:
    distance[i] <- inf
    parent[i] <- -1


start <- start is the starting node
```

```
26
27 # Now doing the first N-1 iterations of bellman ford algorithm
28 for i<- 1 to n-1 do:
29     for edge in edges do:
30         u  = edge.first  # the starting vertex
31         v  = edge.second # the ending vertex, means edge from u to v
32         w = R*[u,v]
33         if (distance[v] >  distance[u] + w) then:
34             distance[v] = distance[u] + w
35             parent[v] = u
36
37
38 # Now doing the Nth iteration to find if we are still able to make any
39 # improvement, if we are able to make an improvement that means the graph
40 # has a negative weight cycle.
41
42 hasNegCycle <- False
43 relaxed_vertex <- -1
44
45 for edge in edges do:
46     u  = edge.first  # the starting vertex
47     v  = edge.second # the ending vertex, means edge from u to v
48     w = R*[u,v]
49     if (distance[v] >  distance[u] + w) then:
50         hasNegCycle <- True
51         relaxed_vertex <- v
52         break
53
54 # if hasNegCycle is True that means we found a negative weight cycle in the
55 # graph. This means there exists a cycle which has product of the R[i,j] of
56 # edges greater than 1 and relaxed_vertex is one of the vertices of the cycle.
57
58 # if relaxed_vertex is -1 then we don't have any negative cycle and we end the
59 # program here
60
61 if (relaxed_vertex =-1) then:
62     End the procedure
63
64 for i<-0  to N-1 do:
65     relaxed_vertex= parent[relaxed_vertex]
66
67 # Now backtracking using the parent array to detect cycle
68 cycle <- []     # empty list that store the vertices of the cycle
69
70 cycle.append(relaxed_vertex)
71 v = parent[relaxed_vertex]
72
73 while True do:
74     cycle.append(v)
75     if (v = relaxed_vertex) then:
76         break
```

```
77      else:
78          v= parent[v]
79
80  cycle = reverse(cycle)  # reversing the vertices present in the cyle
81
82  # We have vertices stored in the cycle vector and if the elements of cycle are
83  # c1, c2, ..., ck, ck+1 = c1 then edges that are included in the graph are
84  # c1->c2, c2->c3,  ..., ck->c1
```

Time Complexity Analysis:
The overall time compelxity of the following algorithm will be $O(n^3)$. Most of the part is same as that of 3a, where we are running the Bellman ford Algorithm.

1. We are storing the parent during each iteration and this takes $O(mn)$ time.
2. When running the n^{th} iteration we need to find the vertex which is a part of the negative cycle. This is done in $O(m)$ time.
3. Next we run the for loop in the lines 70 to 75. This is a whie True loop but it will end as soon as we have encountered the first vertex again and hence this can atmost n times. Thus the time complexity of this part is also $O(n)$.

Hence the overall time compelxity will be $O(mn)$ and since m is $O(n^2)$, the time complexity becomes $O(n^3)$

4 Coin Change

4.1

We have coins of k denominations, and the vector d which stores the values of that denomination. $d[i]$ means the value of the i^{th} currency. Also we have been given an infinite supply of the coins. We need to find the number of ways in which we can use the currency to sum up to an amount of n.

The problem can be solved using dynamic programming. We will assume that since we need to make the amount, it doesn't matter in what order do we pick the coins. That is if we are making amount =3 using 1 and 2 denomination coins, then it doesn't matter if we choose 1 first and then add 2, or we add 2 first and then 1.

4.1.1 Algorithm outline

To make sure that we are not making any repetitions, that is the coins are being considered in unordered manner, we will count the number of ways to make some amount denomination by denomination. First we will see the number of ways to make different amounts from 0 to n using $d[0]$ and store that in a *ways* vector. Next we will update the array using $d[1]$ and so on till $d[k]$.

4.1.2 Algorithm

```
1  d[k] <- this contians the denomination values
2
3  ways[n+1] <-  this contains number of ways we can make a particular amount
4
5  for i <- 0 to n do:
6      ways[i] = 0      # initially no ways to make any amount
7  ways[0] =1           # there is only one way, ie take no coin
8
9  for i <- 0 to k-1 do:
10     for j <- 1 to n do:
11         if (j-d[i] >=0) then:
12             ways[j] += ways[j-d[i]]
13
14 return ways[n]
15 # ways[n] contains the number of ways we can make chang amount n
```

4.1.3 Proof of correctness

Invariant: After i iterations of the outer loop, the ways array stores the total number of ways to form the amounts using the denominations of $d[0], d[1], d[2], ..., d[i-1]$.

The above invariant can be proved using Weak induction on i (the number of denominations we have)

Induction Hypothesis:
After i iterations of the outer for loop, $ways[j] \forall j \in [0, 1, 2, \ldots, n]$ stores the number of ways to form amount j using the denominations from $d[0], d[1], ..., d[i-1]$.

Base Case:
The case when we don't have any coins. It is only possible to form the amount 0 in this case and the number of ways to form the amount 0 is 1, ie use no coin.

Induction step:
In the $(i+1)^{th}$ iteration, we have another possible value of denomination, ie $d[i]$. Now we need to count the number of ways we can form can form the amounts using denominations $d[0], d[1], ..., d[i]$. Now in the inner loop we iterate over all the amount values. After the introduction of this new denomination, if we are using this to make a certain amount, say amnt, the number of ways will be the number of ways to make the amount $(amnt - c * denomination)$ using the first i denominations and c number of d[i] coins (d[i] also because we have infinite supply of these coins). So in the inner loop we iterate in an increasing order of amount and update the ways array. To update ways[amnt] we need to add the number of ways to make (amnt- denomination), (amnt- 2*denomination), ..., (amnt - x*denomination) (till the amount values are greater than zero) using the denominations upto i. Now since in the induction hypothesis we have assumed that the number of ways to form certain amount is stored correctly in the ways array, we can directly add these values of ways[amnt] to get the result. But there is a little optimization, we can keep updating the count array for previous amounts and number of ways of $(amnt - 2 * denomination)$ and so on will get covered in $count[amnt - denomination]$. Thus we only need to add count[amnt- denomination] to update count[amnt]. Doing it in this manner ensures that there will be no repetitions since we have used d[i] for the first time and we are separately counting the number of ways in which we use different number of coins of this denomination.

Hence our Invariant and Induction Hypothesis are correct and after k iterations, ways[n] stores the number of ways to form amount n using infinite supply of coins of denominations $d[0], d[1], ..., d[k-1]$.

4.1.4 Time complexity Analysis

The time complexity of the above algorithm will be $O(nk)$ since we running 2 nested for loops, the outer loop runs for n iterations and the inner loop runs for k iterations. Thus overall the program runs for nm iterations. Thus the time complexity is $O(nk)$

4.1.5 Space Complexity Analysis

The space required is for storing count of coins required for a certain denomination. It needs $O(n)$ space. The other space is required to store the denominations, which need $O(k)$ space. Thus the overall space complexity is $O(n+k)$.

4.2

The problem can be thought in a recursive manner. A certain amount has to be made from some other amount by using more coin. So we can take a minimum over all the possibilities and then consider the possibility in which least number of coins are needed.

4.2.1 Algorithm outline

We will maintain an array count in which we will store the number of coins needed to make a certain amount. Next we know that no coin is needed to make the amount 0 and hence count[0]= 0. Then we iterate for every amount to find the minimum number of coins required to make it.
It is 1+ count[amount- denomination value] and we will take a minimum over all the possible denominations. This ways in the end we will have the number of coins required to make amount n.

4.2.2 Algorithm

```
d[k] <- this contians the denomination values

count[n+1] <- this stores number of coins needed to make certain  denomination

# intitialize every value with -1
for i <- 0 to n do:
    count[i] = INF  # initially denoting it's not possible to form certain amt
count[0] = 0        # No coins are needed to form amount 0

for i <- 1 to n do:
    for j <- 0 to k-1 do:
        if (i - d[j] >= 0) then:
            if (count[i-d[j]] < INF) then:
                count[i] = min(count[i], 1+ count[i-d[j]])

return count[n]
# this denotes minimum no. of coins required to form a certain amount
```

4.2.3 Proof of correctness

Invariant: After every iteration of the outer loop, count[i] stores the minimum number of coins which are required to make amount i.

The above invariant can be proved using Strong induction on i, the amount

Induction Hypothesis:
After i iterations of the outer for loop, $count[i] \forall i \in [0, 1, 2, \ldots, i]$ stores the minimum number if coins that are required to form amount i.

Base Case:
The case when the amount is zero. Since no coin is required to form amount 0, hence the minimum number of coins required to form 0 are 0

Induction step:
In the $(i+1)^{th}$ iteration we need to find the minimum number of coins required to form amount i+1. This amount has to be made by using some coins. In the inner loop we go through all the possible denominations. For a particular denomination if we want to make the amount i+1, then the minimum number of

coins required will be (1+ min coins for (i+1- denomination)), and since (i+1-denomination) is less than i+1, and using induction hypothesis we know count[i+1- denomination] already stores the minimum number of coins required to form it. Hence the minimum number of coins required to form amount i+1 using a particular denomination will be $1 + count[i + 1 - denomination]$. Since it has to form through some of these k denominations only, so we can take a minimum over all the possible denomination by iterating over all denominations in the inner for loop. Which ever denomination results in overall less number of coins is accepted and stored. Hence after i+1 iterations count[i+1] stores the minimum number of coins required to form the amount i+1.

Hence our Invariant and Induction Hypothesis are correct and after n iterations, count[n] stores the minimum number of coins required to form amount n.

4.2.4 Time complexity Analyssis

The time complexity of the above algorithm will be $O(nk)$ since we running 2 nested for loops, the outer loop runs for n iterations and the inner loop runs for k iterations. Thus overall the program runs for nm iterations. Thus the time complexity is $O(nk)$

4.2.5 Space Complexity Analysis

The space required is for storing count of coins required for a certian denomination. It needs $O(n)$ space. The other space is required to store the denominations, which need $O(k)$ space. Thus the overall space complexity is $O(n + k)$.