Indian Institute of Technology Delhi

COL351 Analysis and Design of Algorithms: Assignment 2

Ishita Chaudhary: 2019CS10360

Mohammed Jawahar Nausheen: 2019CS10371

Contents

Algorithms Design Group

The problem requires an optimal partition of C[1,2,..n] into 3 subsets S_1, S_2, S_3, i.e, $S_1 \cup S_2 \cup S_3 = C$ and S_1, S_2, S_3 are mutually disjoint, for which max$(sum(S_1), sum(S_2), sum(S_3))$ is minimum. Note that once S_1 and S_2 are fixed, S_3 can be uniquely determined. Thus we focus on finding all possible combinations of S_1, S_2 using a 3 dimensional dp array which denotes whether such a combination is possible.

Claim 1: dp[i][s_1][s_2] $= 1 \iff \exists S_1, S_2 \subseteq C[1,...i], S_1 \cap S_2 = \phi$, $sum(S_1) = s_1 \wedge sum(S_2) = s_2$
Proof is by induction on i
Base case: When i = 0, only zero sum is possible, hence dp[0][s_1][s_2] = 1 when s_1, s_2 are both 0 and 0 otherwise.
Induction Hypothesis: The claim holds true $\forall 0 \leq k < i$
Induction Step: dp[i][s_1][s_2] = 1 in one of the following ways.
Case 1: dp[i-1][s_1][s_2] = 1
From IH, $\exists S'_1, S'_2 \subseteq C[1,...i-1], S'_1 \cap S'_2 = \phi$, $sum(S'_1) = s_1 \wedge sum(S'_2) = s_2$, thus S'_1, S'_2 are also subsets of $C[1,..i]$. The claim holds.
Case 2: dp[i-1][s_1 - C[i]][s_2] = 1
Again from IH, $\exists S'_1, S'_2 \subseteq C[1,...i-1], S'_1 \cap S'_2 = \phi$, $sum(S_1) = s_1 - C[i] \wedge sum(S_2) = s_2$. Now $S_1 := S'_1 \cup \{C[i]\}$, $S_2 := S'_2$ can be constructed.
Case 3: dp[i-1][s_1][s_2- C[i]] = 1
Similar to case 2, we construct $S_1 := S'_1$, $S_2 := S'_2 \cup \{C[i]\}$.
Since dp[i][s_1][s_2] is Or of these three values, the argument is exhaustive. Also when dp[i][s_1][s_2] = 0 then assume to the contrary $\exists S_1, S_2$.
Case 1: C[i] $\notin S_1$, C[i] $\notin S_2$. Then S_1, S_2 ae subsets of C[1,...i-1] which implies dp[i-1][s_1][s_2] is 1, a contradiction.
Case 2: C[i] $\in S_1$, C[i] $\notin S_2$. Then $S_1 - \{C[i]\}$, S_2 are subsets of C[1,..i-1] implying dp[i-1][s_1-C[i]][s_2] is 1, again a contradiction.
Case 3: C[i] $\notin S_1$, C[i] $\in S_2$. Again similar to case 2, $S_1, S_2 - \{C[i]\}$ are subsets of C[1,..i-1], dp[s_1][s_2 - C[i]] is 1, a contradiction.

Once possible subset combinations are identified, we can iterate over dp[n] to find the min of max possibility required. Then we construct the sets S_1, S_2, S_3 using the cases in the above claim's proof. When $C[i] \notin S_1, S_2$, $C[i] \in S_3$ and S_1, S_2 are the same whose existences we proved above.

Time and Space complexity analysis: It is given that the number of exercises in each chapter is bounded by the number of chapters. Thus, sum $\leq n^2$, where n is the number of chapters or size of C. The first for loop initializing dp[0] takes $sum * sum$ time thus takes $O(n^4)$ time. The next for loop takes $n * sum * sum$ time, thus it is of $O(n^5)$. The next loop is also of $O(n^4)$ and the final loop that goes from i = n to 0 takes O(n) time. Thus total time complexity of the algorithm is $O(n^5)$ and in terms of sum it is $O(n * sum^2)$. The space complexity (size of dp array) is also $O(n^5)$ or $O(n * sum^2)$.

```
Function minMaxPartition(chapters C, size n):
    sum:= 0
    forall c ∈ C do  sum = sum + c
    initialize dp[n + 1][sum + 1][sum + 1]
    forall s1 = 0 to sum do
        forall s2 = 0 to sum do
            if s1 = 0 ∧ s2 = 0 then dp[0][0][0] = 1
            else dp[0][s1][s2] = 0
        end
    end
    forall i = 1 to n do
        forall s1 = 0 to sum do
            forall s2 = 0 to sum do
                dp[i][s1][s2] = dp[i-1][s1][s2]
                if s1 ≥ C[i] then  dp[i][s1][s2] = dp[i][s1][s2] ∨ dp[i][s1 -C [i]][s2]
                if s2 ≥ C[i] then  dp[i][s1][s2] = dp[i][s1][s2] ∨ dp[i][s1][s2 -C [i]]
            end
        end
    end
    answer:= sum
    forall s1 = 0 to sum do
        forall s2 = 0 to sum do
            if dp[n][s1][s2] then
                if  max(s1, s2, sum − s1 − s2) < answer then
                    answer = max(s1, s2, sum − s1 − s2))
                    pair = (s1, s2)
                end
            end
        end
    end
    S1, S2, S3 := φ
    i := n, (s1, s2) := pair
    while i ≠ 0 do
        if dp[i-1][s1][s2] then  S3.insert(C [i])
        else if s1 ≥ C[i]∧ dp[i-1][s1 - C [i]][s2] then
            S1.insert(C [i])
            s1 = s1 - a[i]
        end
        else if s2 ≥ C[i]∧ dp[i-1][s1][s2-C [i]] then
            S2.insert(C [i])
            s2 = s2 - a[i]
        end
        i = i - 1
    end
    return S1, S2, S3
```

Course Planner

2.1 Ordering of Courses to Take All n Courses

The problem can be represented as a directed graph where $(w, v) \in G \iff v \in P(w)$.
Claim 1: A valid ordering of the courses is possible iff G is acyclic.

Consider a cycle $C \in G$. For any vertex $v \in C \exists w \in C$, s.t (v,w) $\in C$. Assume that we choose to do course v first ,without loss of generality. Then from the previous claim $\exists$ edge (v,w) which implies $w \in P(v)$, thus v cannot be done before w. The same can be said for any vertex of the cycle and thus no valid ordering exists for G if G contains a cycle.
Now to show the other direction of the claim, i.e, if G is acyclic then a valid ordering exists, we show the existence of an ordering using the topoogical order of a DAG.

Claim 2: A topological ordering of DAG G is a valid course ordering.
In a topological ordering all edges go in the same direction. Consider an ordering in which all the edges go from right to left. Since there is an edge from w to each $v \in P(w)$, all prerequisites of w are ordered before w, complying with the condition required.

Hence finding a topological sort of the DAG G, where all edges go from right to left solves the problem. There are two ways this can be done. One is ordering the vertices according to their DFS finish times, and the other is Kahn's algorithm of choosing the first vertex and reducing the graph to a sub-graph. Here, we proceed with the latter.

Function `TopologicalSort(`*courses C, prerequisite mapping P*`)`**:**
  forall $c \in$ C **do** inDegree (c) := 0
  forall $c \in$ C **do**
    forall $p \in$ P(c) **do** inDegree (p) $\leftarrow$ inDegree (p) + 1
  end
  forall $x \in$ C **do**
    if inDegree(x) = 0 **then** queue.push(x)
  end
  ordering = <>
  while queue $\neq \phi$ **do**
    $v \leftarrow$ queue.*pop*()
    ordering $\leftarrow$ ordering.v
    forall $x \in$ P(v) **do**
      inDegree(x) $\leftarrow$ inDegree(x) $-$ 1
      if inDegree(x) = 0 **then** queue.push(x)
    end
  end
  if ordering.*size*() $\neq$ |C| **then** **return** <>
  return *reverse(*ordering*)*

Claim 3: In the *ordering* produced by the algorithm, all edges go from left to right.

Let v be a vertex in G whose indegree is 0, v is added first, and thus placed at the start of topological ordering produced by the algorithm. Now there cannot be an edge from the right to v since its indegree is 0. The problem can be safely reduced to v.$<ordering(G-v)>$, thus reducing the indegrees of all neighbours of v is valid.
Since we require an ordering of the courses where each edge goes from right to left, we return

the reverse of *ordering*.

Claim 4: The algorithm returns <> iff G contains a cycle.
If G contains a cycle C $v_1, v_2, ...v_k, v_1$, the indegree of each vertex v_i in C is non-zero in G and any sub-graph of G containing C, thus v_i will never be added to the queue and *ordering*. Thus size(ordering) is less than $|G|$.
Now suppose a set of vertices whose indegrees are non-zero when the algorithm ends. Starting from v_1, make a sequence by adding a vertex that has an edge to v_1 and repeat. Since the indegrees are non-zero we can always extend the sequence and at some point a vertex will be repeated thus proving existence of a cycle.

Time and space complexity analysis: Calculating indegrees takes O(m+n) time. The queue is filled and popped a maximum of O(n) times and the other for loop reduces indegree of a vertex by one in each iteration. So it runs a total of O(m) times. The time complexity of the algorithm is O(m+n). An array of indegrees and the queue are of size n so the space complexity is O(n).

2.2 Minimum Number of Semesters to Complete All n Courses

From 2.1 we know that an ordering is possible only when G is acyclic. Hence in presence of a cycle the number of semesters is assumed to be INF, since any number of semesters cannot be used to provide a valid ordering.
For a DAG G, we shall show that the minimum number of semesters required to do the courses say M(G), is equal to the longest path of in G, say L. We use the following two claims for this.

Claim 1: L $\leq$ M(G)
Let P be the longest path in G. Then we require a minimum of $|P|$ semesters to finish the courses, since no two courses in the path can be done together.

Claim 2: If an optimal course allotment is of size $M(G)$ semesters, a path exists in G that is of length $M(G)$.
Let C_k denote the courses done in semester k according to an optimal course allotment with number of semesters M(G). Then for each edge (w, v), $w \in C_i, v \in C_j, j < i$ according to the definition of G in 2.1.
Let $S_k \subseteq C_k$ be defined as $S_k = \{w \in C_k | \exists v \in C_{k-1}, (w, v) \in G\}$
Proof is by construction. We iteratively modify the allotment without changing its validity or size using the following observation. If $v \in C_k \wedge v \notin S_k$ then v can be shifted to S_{k-1} instead.Thus we iteratively do $C_k := S_k$ and $C_{k-1} := C_{k-1} \cup (C_k \backslash S_k)$ for $k = M(G) to 1$. Note that the $S_k \neq \phi$ because $\exists v \in C_k$ having an edge to C_{k-1} since otherwise, C_k and C_{k-1} could be clubbed together and the optimality of M(G) is violated. Hence each C_k is non-empty even after the iterative modification.
Now after the iteration ends, we have $\forall k > 1, \forall v \in C_k, \exists w \in C_{k-1}$ s.t, $(v, w) \in G$. Thus $\exists$ path $a_m.a_{m-1}.a_{m-2}....a_1$ where $a_i \in C_i$ and m = M(G).
Now since L is the length of the longest path in G, from claim 2 it follows that M(G) $\leq$ L. Finally using claim 1, M(G) = L.

Claim 3 : Let $v_1, v_2, ...v_n$ be the sequence in which vertices were pushed into the queue. Then at iteration i, dp[v_j] contains the longest path that ends at $v_j \forall j \in [1, i]$
Base case: Consider a vertex v of indegree 0. There is no incoming edge to v and thus no path to v. dp(v) is initialised at 0 thus the claim holds true. Also v $\notin$ P(w) for any w, thus dp(v) stays 0. Hence the claim holds true for i= 1.
Induction Hypothesis: The claim holds true for i = k
Induction Step: From 2.1 we know that the sequence $v_1, ..v_n$ is a topological ordering. Thus there cannot be any edges from right to left in this case implying, $v_i \notin P(v_j)$ where i < j. Thus dp[$v_1, ...v_k$] stays correct. Now the longest path that ends at v_{k+1} is max(1 + dp[w]) where $v_{k+1} \in P(w)$.Let w = v_i, then $i \in [1, ..k]$ from the property of topological ordering. Also at iteration i, dp[v_i] contains length of longest path ending at v_i from IH and dp[v_{k+1}] is derived correctly.

Time and space complexity: The first loop takes O(n) time while the second for loop takes O(m+n) time. The while loop can run a maximum of n times since in each iteration, a vertex is popped from the queue, and its neighbours are visited in this iteration. Thus, this loop also takes O(m+n) time. Finally the last for loop takes O(n) time. The time complexity of the algorithm is O(m+n).

```
Function LongestPath(courses C, prerequisite mapping P):
    forall c ∈ C do
        inDegree (c) := 0
        dp (c) := 0
    end
    forall c ∈ C do
        forall p ∈ P(c) do  inDegree (p) ← inDegree (p) + 1
    end
    forall x ∈ C do
        if inDegree(x) = 0 then queue.push(x)
    end
    count:= 0
    while queue ≠ ϕ do
        v ← queue.pop()
        count ← count + 1
        forall x ∈ P(v) do
            dp (x) ← max(dp (x), 1 + dp (v))
            inDegree(x) ← inDegree(x) − 1
            if inDegree(x) = 0 then  queue.push(x)
        end
    end
    if count ≠ |C| then return INF
    ans := 0
    forall x ∈ C do ans ← max(ans, dp(x))
    return ans
```

2.3 All Pair of Courses with Disjoint Set of Pre-requisites

For this problem, we shall represent the courses as a graph G' where edge $(v, w) \in G' \iff v \in P(w)$. Now $L(v) := \bigcup_{w \in P(v)} \{w\} \cup L(w)$, i.e, L(v) is the set of proper ancestors of v in the graph G'. We wish to find all pairs (a,b) for which L(a) ∩ L(b) = ϕ.

Claim 1: In a DFS tree T rooted at r, all pairs (v,w) where v,w ∈ T-r, $L(v) \cap L(w) \neq \phi$. Since v ∈ T and $v \neq r$, r is an ancestor of v. Thus $r \in L(v)$. The same can be said about w, thus $r \in L(v) \cap L(w)$ and therefore $L(v) \cap L(w) \neq \phi$.

Claim 2: Let v,w ∈ G' and $r \in L(v) \cap L(w)$ then any DFS tree rooted at r will visit v,w. We know that a DFS tree rooted at r visits all vertices reachable from r. We shall therefore show that v and w are indeed reachable from r using the construction of L(v) and L(w).
$r \in L(v) \implies r \in P(v) \vee (\exists s \in P(v), r \in L(s))$
If $r \in P(v)$, there exists edge $(r, v) \in G'$, thus v is reachable from r. Otherwise we use L(s) to construct a path from r to s and join it with edge (s,v) making a path from r to v. v is again reachable from r.

From claim 1 and 2 it is clear that if $L(v) \cap L(w) \neq \phi$, then v and w will be a part of the same DFS tree rooted at any $r \in L(v) \cap L(w)$. From the converse it follows that if $L(v) \cap L(w) = \phi$, v and w will never occur as non-root vertices of any DFS tree. Thus we run DFS from each vertex in G' and mark all pairs of vertices that are visited in the tree as false. In the end, all remaining unmarked vertices have $L(v) \cap L(w) = \phi$.

Time and space complexity analysis Getting successors S from prerequisites P can be done in $O(m + n)$ time. The next loop initializes matrix $IntersectionEmpty$, which is done in n^2 time. In the following block, the outermost loop runs n number of times. DFS from each vertex takes O(m+n) time complexity and extracting all possible pairs from the list produced by DFS takes $O(n^2)$ time. Thus each of the n iterations takes $O(m + n) + O(n^2)$ time. Also, $m = O(n^2)$ hence the time taken by the outer for loop is $n * n^2$, i.e, $O(n^3)$. The last loop is of $O(n^2)$. Thus, the algorithm takes $O(n^3)$ time and $O(n^2)$ space.

```
Function DFS(vertex v, edge list S, visited array vis, list R):
    vis[v] = true
    R.add(v)
    forall w ∈ S(v) do
        if (not vis[w] then DFS (w, S, vis, R)
    end
    return R

Function FindAllPairs(courses C, prerequisite mapping P):
    forall c ∈ C do
        forall p ∈ P(c) do  S(p) = c

    end
    forall v, w ∈ C do
        IntersectionEmpty [v][w] = true
        IntersectionEmpty [w][v] = true
    end
    forall c ∈ C do
        V = DFS (c) - c
        forall v ∈ V do
            forall w ∈ V do
                IntersectionEmpty [v][w] = false
            end
        end
    end
    forall v ∈ C do
        forall w ∈ C do
            if IntersectionEmpty [v][w] then  PairSet.insert((v,w))
        end
    end
    return PairSet
```

Forex Trading

Given a trader aiming to make money by taking advantage of price differences between different currencies. The currency exchange rates are modeled as a weighted network, wherein, the nodes correspond to n currencies c_1, ..., c_n, and the edge weights correspond to exchange rates between these currencies. In particular, for a pair (v_i, v_j), the weight of edge (v_i, v_j), say R(i, j), corresponds to total units of currency c_j received on selling 1 unit of currency c_i.

3.1 Existence of a Cycle Over Which Exchanging Money Results in Positive Gain

We have to design an algorithm to verify whether or not there exists a cycle (c_{i1}, ..., c_{ik}, c_{ik+1} = c_1) such that exchanging money over this cycle results in positive gain, or equivalently, the product R[i_1, i_2]xR[i_2, i_3]x...xR[i_{k-1}; i_k]xR[i_k; i_1] larger than 1.

From above condition, the below follows:

- R[i_1, i_2]xR[i_2, i_3]x...xR[i_{k-1}; i_k]xR[i_k; i_1] > 1
- (1/R[i_1, i_2])(1/R[i_2, i_3])...(1/R[i_{k-1}; i_k])(1/R[i_k; i_1]) < 1
- Take log both sides to get,
- log((1/R[i_1, i_2])(1/R[i_2, i_3])...(1/R[i_{k-1}; i_k])(1/R[i_k; i_1])) < 0
- log((1/R[i_1, i_2])) + log((1/R[i_2, i_3]))+...+log((1/R[i_{k-1}; i_k])) + log((1/R[i_k; i_1])) < 0
- (-log(R[i_1, i_2])) + (-log(R[i_2, i_3]))+...+(-log(R[i_{k-1}; i_k])) + (-log(R[i_k; i_1])) < 0

Now, alternatively define the weight of an edge (v_i, v_j) as wt(v_i, v_j) = -log(R[i, j]). Our problem is now boiled down to verify if there exists a negative weight cycle in a directed weighted graph, where edge weights are denoted by wt(v_i, v_j) for edge (v_i, v_j). We can use Bellman-Ford Algorithm to do the same. For this, we need a source vertex v_0.

Since it is given that for each pair (v_i, v_j) the exchange rate is R(i, j), the modeled graph G= (V, E) has an edge between all pair of vertices, hence, |V|= n and |E|= ${}^nC_2$.

Consider the graph G'= (V+v_0, E'), where E'= E + (v_0, v_i) $\forall i \in [1, n], such that wt$($v_0$, v_i) = 0. The graph G' will not have additional cycles as compared to G, as the new edges added have v_0 as the source vertex and no such edge exists which has v_0 as its destination vertex. And since the new vertex v_0 added has an edge to all existing vertices, each negative weighted cycle is reachable from source vertex v_0. Run Bellman-Ford Algorithm on G'.

Claim 0: Let d[v, i] denote the shortest path from source vertex v_0 to v that uses at most i edges. Then d[v,i] = min(d[v,i-1], d[u,i-1] + weight(u,v)) for all edges (u,v) $\in$ G'.
The claim can be proved by induction. Assume that the shortest path of at most i edges contains less than i edges then d[v,i] = d[v,i-1] is minimum over all such paths. Otherwise if it is an i length path length, then let the path be v_0,...u,v. The path between v_0 and u uses less than i edges and thus its length is d[u,i-1]. Adding the edge (u,v) gives the shortest path length with at most i edges.

Claim 1: After the ith iteration of for loop distance[v] $\leq$ d[v,i].

Function `BellmanFord`(*graph G', src vertex* v_0):
- **forall** $v_i \in$ VertexSet **do** distance $[v_i]$:= infinite
- distance $[v_0]$:= 0
- //relax all edges |v|-1 times
- **forall** $i \in [1, |\text{VertexSet}| - 1]$ **do**
  - **forall** *edge* $\in$ EdgeSet **do**
    - vertex src= G'.EdgeSet [edge].source
    - vertex dst= G'.EdgeSet [edge].destination
    - weight w= G'.EdgeSet [edge].weight
    - **if** distance$[src]$ $\neq$*infinite and src + w* $\leq$ distance$[dst]$ **then** distance [dst] = distance [src] + w
  - **end**
- **end**
- **forall** *edge* $\in$ EdgeSet **do**
  - vertex src= G'.EdgeSet [edge].source
  - vertex dst= G'.EdgeSet [edge].destination
  - weight w= G'.EdgeSet [edge].weight
  - **if** distance$[src]$ $\neq$*infinite and src + w* $<$ distance$[dst]$ **then return** *True*
- **end**
- **return** *False*

Proof: Proof by induction on i.
Base case: For base case consider i=1. Here, d[v, 1] denote the shortest path from v_0 to v that uses at most one edge, i.e., neighbours of v_0. In each iteration, we consider all edges, so after one iteration, distance[v] will be equal to d[v, 1]. Note that distance[v] can use an updated value of some distance[v], however this happens only if this path is of lesser cost. Hence, distance[v] $\leq$ d[v,1]
Induction hypothesis: Assume the claim holds true after k iterations. That is, distance[v] $\leq$ d[v, k].
Induction Step: Let $v_0, ...v$ be the shortest path from v_0 to v that uses at most k+1 edges. If the path contains less than k+1 edges then, d[v,k+1] = d[v,k] and hence distance[v] $\leq$ d[v,k+1], since distance[v] is reduced throughout the algorithm.
If the path contains k+1 edges then in the (k+1)th iteration, for some vertex u, distance[v] $<$ distance[u] + weight(u,v). Now distance[u] $\leq$ d[u,k], thus distance[v] $\leq$ d[v,k+1] from claim 0.

Claim 2: The above algorithm detects negative cycle. That is, there exists a negative cycle reachable from source vertex v_0, iff for some edge (v_i, v_j), distance[v_j] $>$ distance[v_i] + wt(v_i, v_j), in the $(|VertexSet|)^{th}$ iteration.
Proof: Proof by contradiction.
After $|VertexSet| - 1 = k$ iterations, we know that distance[v_j] $\leq$ d[v_j,k]. If distance[v_j] $>$ distance[v_i] + wt(v_i, v_j) in the (k+1)th iteration, we know that the path uses (k+1) or more edges. Since any simple path can only use k edges, the path from v_j to v_0 contains a cycle. If the cycle weight was non-negative, we could safely remove it to achieve the same or lesser path cost. But since this minimal path was possible only with more than k+1 edges, weight of the cycle is negative.
Now suppose that the graph contains a negative cycle but there is no relaxation in the $(k+1)^{th}$ iteration. Let $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_k \rightarrow v_1$ is a negative cycle reachable from v_0. Implying, wt(v_k, v_1) $+\sum_{i=1}^{k-1}$ wt(v_i, v_{i+1}) $<$ 0.
Assume distance[v_j] $\leq$ *distance*[v_i] + wt(v_i, v_j) for all edges (v_i, v_j) in the cycle.
Sum up the above inequality for all vertices in the cycle. The sum of distance[v_j] will be equal to

sum of distance[v_j] over all vertices, as each vertex is a source vertex and a destination vertex. Hence, we finally get
$\implies 0 \leq \sum_{i=1}^{k}$wt($v_i$, v_j)
$\implies 0 \leq$ wt(v_k, v_1) $+\sum_{i=1}^{k-1}$ wt(v_i, v_{i+1})
This contradicts our assumption that $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_k \rightarrow v_1$ is a negative cycle.

Analysis of Time Complexity: Since the graph has |V|= n and |E|= $^nC_2$= O(n^2). Building the graph takes O(|V| + |E|) time, i.e., O(n + n^2) = O(n^2). In Bellman-Ford algorithm, first for loop which initializes values of distance array iterates over the vertex set and is O(|V|). The next we have are two nested for loops, iterating over vertex set and edge set respectively, hence it comes out to be O($|V||E|$). The final for loop, iterates over the edge set, and hence is O(|E|). The complexity of the Bellman-Ford algorithm comes out to be O(|V| + $|V||E|$ + |E|) = O(n + n^3 + n^2) = O(n^3).

To build the graph and run the Bellman-Ford algorithm takes O(n^2 + n^3) = O(n^3) time, or in terms of edges m, $O(mn)$.

3.2 Cubic Time Algorithm to Print a Negative Weighted Cyclic Sequence in a Directed Graph

The algorithm in the previous part detects if there exists such a cycle or not, in this part we want to print the cycle, if it exists. In previous part, if we perform the $|V|_{th}$ iteration of Bellman-Ford, i.e., relax edges |V| times, and choose a vertex (source or destination) of the edge which is relaxed in this iteration. If no edge is relaxed in this iteration, there exists no negative weighted cycle in the graph.

Use this chosen vertex to print the negative cycle, using its ancestors, until we reach the same vertex.

Claim 1: If check is not empty, then the vertex labeled as check is a part of negative weighted cycle.
Proof: Consider $|V|$ = n. Initially distance[v] is infinite ∀ v ∈ V. If vertex v is reachable from the source vertex v_0, then there must exist an acyclic path from v_0 to v. The number of edges in shortest path from v_0 to any other vertex can be at most n-1. We relax all edges (n-1) times, which guarantees that each edge in the shortest path from v_0 to v must have been relaxed, so that distance[v] ≠ infinite for any v ∈ V. Note that, after these (n-1) iterations, distance[v] denote the length of shortest path from source vertex v_0 to v, if there exists no negative cycles in the graph.
Consider the next iteration, where all edges are relaxed one more time. If there exists no negative cycle, then for all edges distance[edge.source] + weight(edge) ≥ distance[edge.destination]. In case this condition isn't satisfied, we can imply that the edge is a part of a negative cycle, and hence check vertex (which is updated to the destination vertex of corresponding edge), also lies in the negative cycle.

Claim 2: The above algorithm prints the negative cycle, if it exists.
Proof: As we proved in the previous subsection (click here), that the above algorithm detects a negative cycle, if it exists.
We have to verify if it prints the negative cycle correctly. Let $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_k \rightarrow v_1$ is a negative cycle. After $|VertexSet|$ -1 relaxations, we check for the existence of negative cycle in next iteration. If it exists, the check is updated to one of the vertices of the cycle. Let check = v_i for some i ∈ [i,k].
We initialise an array cycle, which will maintain the order of vertices in cycle. Since, we have maintained parent for each vertex, we will iterate from vertex to its parent until we get back to our initial vertex check.
Without loss of generality, consider i=1, i.e., check = v_1. Parent of v_1 is v_k. Hence, the array cycle obtained will be of the form, $v_1, v_k, v_{k-1}, ..., v_2$. This array is in the reverse order of the direction of cycle, because we iterated from vertex to its parent. We will print the array from last index to first, to get the the correct order of cycle.
The printed order will be $v_2, v_3, ..., v_k, v_1$, which is the desired result. The above algorithm successfully returned the negative cycle.

Time and space complexity analysis: Since the graph has |V|= n and |E|= ${}^{n}C_2$= O(n^2). Building the graph takes O(|V| + |E|) time, i.e., O(n + n^2) = O(n^2). To print negative cycle, we use Bellman-Ford algorithm, first for loop which initializes values of distance array iterates over the vertex set and is O(|V|). The next we have are two nested for loops, iterating over vertex set and edge set respectively, hence it comes out to be O($|V||E|$). The next for loop, iterates over the edge set, and hence is O(|E|). If there exists a negative cycle, the while loop will terminate after (length of cycle) iterations, the length of cycle is bound by O(|V|). Similarly, to print a cycle, it wold iterate over (number of vertices in cycle) iterations, number of vertices

```
Function PrintNegativeCycle(graph G, src vertex v_0):
    forall v_i ∈ VertexSet do distance [v_i] := infinite, parent [v_i] := -1
    distance [v_0] := 0
    //relax all edges |v|-1 times
    forall i ∈ [1, |VertexSet| − 1] do
        forall edge ∈ EdgeSet do
            vertex src= G.EdgeSet [edge].source
            vertex dst= G.EdgeSet [edge].destination
            weight w= G.EdgeSet [edge].weight
            if distance[src] ≠infinite and distance [src] + w ≤ distance[dst] then
                distance [dst] = distance [src] + w
                parent [dest]= src
            end
        end
    end
    check := empty
    forall edge ∈ EdgeSet do
        vertex src= G.EdgeSet [edge].source
        vertex dst= G.EdgeSet [edge].destination
        weight w= G.EdgeSet [edge].weight
        if distance[src] ≠infinite and distance [src] + w < distance[dst] then
            check = dst
            parent [dst]= src
            break
        end
    end
    if check ≠ empty then
        cyclicSeq := empty
        currentVertex := check
        while currentVertex ∉ cyclicSeq do
            cyclicSeq.push(currentVertex)
            currentVertex=parent [currentVertex]
        end
        print curentVertex
        //since we iterated from a vertex to its parent vertex
        while cyclicSeq.last() ≠ currentVertex do
            print cyclicSeq.removeLast()
        end
        print currentVertex
    end
    else print No Negative Cycle Found
```

are bounded by O(|V|). The complexity of the Bellman-Ford algorithm (print negative cycle) comes out to be $O(|V| + |V||E| + |E| + |V| + |V|) = O(n + n^3 + n^2 + 2n) = O(n^3)$.

To build the graph and run the Bellman-Ford algorithm takes $O(n^2 + n^3) = O(n^3)$ time. Hence, we have achieved $O(n^3)$ algorithm, or in terms of edges, an O(mn) algorithm. Additional space utilised is the size of distance array which is of O(n).

Coin Change

Given a set of k denominations, d[1], d[2],..., d[k], and a sum of Rs.n.

4.1 Number of Ways to Make Change for Rs.n from Given Denominations

We have to device a polynomial time algorithm to count the number of ways to make change for Rs.n,given an infinite amount of coins/notes of denominations, d[1], . . . , d[k].

Let d=d[1], d[2],..., d[k]. We define a function *TotalNum*, which takes d, n (sum of money to be divided into denominations) and k (types of denominations available) as its parameters and return the number of ways n can be divided according to d.

Claim: *TotalNum(d, n, k) = TotalNum(d, n, k-1) + TotalNum(d, n-d[k], k)*

Proof: Consider the solution set S, which contains all unique possible denominations which sum to n. We have to find |S|. Note that each element belonging to this set S falls in either of the two categories:

S_1 is the solution set which doesn't contain the i^{th} coin.
S_2 is the solution set in which each solution contains at least one i^{th} coin.
For some i, $1 \leq i \leq k$.
Let i= k. Thus $|S_1|$ can be expressed as *TotalNum(d, n, k-1)*. Since, all the solutions in set S_2 contain atleast one k^{th} coin, if we remove one k^{th} coin from each solution in S_2, we reach the problem to compute total number of ways to make change for Rs.*n-d[k]*,given an infinite amount of denominations, d[1], . . . , d[k]. Hence, $|S_2|$ can be expressed as *TotalNum(d, n-d[k], k)*.
Observe that $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \phi$. Implying, $|S| = |S_1| + |S_2|$. Hence, *TotalNum(d, n, k) = TotalNum(d, n, k-1) + TotalNum(d, n-d[k], k)*.

From above reasoning (consider i = k), it can be concluded that *TotalNum(d, n, k)* can be written as the sum of *TotalNum(d, n, k-1)* and *TotalNum(d, n-d[k], k)*. Hence, this problem has optimal substructure and can be solved by dynamic programming.

Claim : After $(i,j)^{th}$ iteration of the for loop, total[i][j] = *TotalNum(d,i,j)*
The claim can be easily proved using the above recursion we showed. Inducting on i + j, in the base case i = 0 and j = 0. total[i][j] is initialised to be 1 which is equal to *TotalNum(d,i,j)*.

Assuming the claim is true for $i+j < p$,
When $j \geq 1$, $i \geq d[j]$,
total[i][j] = total[i][j-1] + total[i-d[j]][j] = *TotalNum(d,i,j-1) + TotalNum(d,i-d[j],j) = TotalNum(d,i,j)*, using the hypothesis and above claim.
If $j < 1$, $i \geq d[j]$,
total[i][j] = 0 + total[i-d[j]][j] = *TotalNum(d,i,j-1) + TotalNum(d,i-d[j],j) = TotalNum(d,i,j)*
If $j \geq 1$, $i < d[j]$,
total[i][j] = total[i][j-1] + 0 = *TotalNum(d,i,j-1) + TotalNum(d,i-d[j],j) = TotalNum(d,i,j)*
If $j < 1$, $i < d[j]$,
total[i][j] = 0 + 0 = *TotalNum(d,i,j-1) + TotalNum(d,i-d[j],j) = TotalNum(d,i,j)*

Time and space complexity analysis: The first for loop, which initialises some entries of the two-dimensional table to one takes O(k) time. The next for loop is nested with another for loop, the first one iterates over n and the next one over the number of denominations available, i.e., k. Hence, the complexity of the nested loops are O(nk). The overall time complexity of

```
Function TotalNum(denominations d, sum n, available denominations k):
    initialise total [n+1][k]
    forall i ∈ [0, k − 1] do
        total [0][i]:= 1
    end
    forall i ∈ [1, n] do
        forall j ∈ [0, k − 1] do
            initialise inc, exc := 0
            if j ≥ 1 then exc ← total [i][j − 1]
            if i ≥ d[j] then inc ← total [i − d[j]][j]
            total [i][j] ← inc + exc
        end
    end
    return total [n][k-1]
```

the suggested algorithm is O(nk + n), which is O(nk). We have achieved a polynomial time complexity. The size of array *total* is (n+1)*k. Hence the space complexity is also of O(nk).

4.2 Finding Change of Rs.n Using Minimum Number of Coins

We need to device an algorithm to find a change of Rs.n using the minimum number of coins from given denominations, d=(d[1], d[2],..., d[k]).
Claim: The above problem can be solved recursively, and modeled as:
minCoins(sumOfMoney) = 0 (if sumOfMoney = 0)
minCoins(sumOfMoney) = min(1 + minCoins(sumOfMoney - d[j])), where j $\in [1, k]$ and d[j] $\leq$ sumOfMoney
Proof: Let minCoins(n) be the minimum number of coins of denominations d[1], d[2],..., d[k] required to make change of n.
Case 1: n can be formed using the given denominations
If the amount n can be formed using the given denominations, in the optimal solution, there exists at least one coin d[i]: d[i] $\leq$ n, for some i $\in [1, k]$.

> **Claim:** The optimal solution of minCoins(n) be p, then p-1 will be the optimal solution of minCoins(n-d[i]).
> **Proof:** Proof by contradiction.
> Consider the optimal solution of minCoins(n). We will divide this solution into two parts, one which contains d[i] and another which sums to n-d[i].
> Assume the second part is not an optimal solution of minCoins(n-d[i]). This implies there is a better solution for minCoins(n-d[i]), say r, such that r < p-1.
> This further implies, r+1 (< p) is the optimal solution for minCoins(n).
> This contradicts our given information that p is the optimal solution of minCoins(n).

From above, we can conclude that if d[i] exists in optimal solution, for some i $\in [1, k]$, minCoins(n)= 1+ minCoins(n-d[i]).
Since, we don't know the value of i, we should check all possibilities and the minimum value will be our optimal solution. Hence, minCoins(n) = min(1 + minCoins(n-d[i])), where i $\in [1, k]$ and d[i] $\leq$ n.
Case 2: n cannot be formed using the given denominations
In a similar fashion as above, we can prove that if n cannot be formed using denominations d[1,..k] then for any d[i]$\leq$n, n-d[i] also cannot be formed using d[1,..k]. Since if it can, then coin d[i] can be added to the change to achieve sum n.
When solved using the above problem, if we draw the recursion tree we can observe that the same nodes appear multiple times. Hence, it has overlapping subproblems and can be solved using dynamic programming.

Time and space complexity analysis: The first for loop, which initialises all entries of the table to infinite takes O(n) time. The next for loop is nested with another for loop, the first one iterates over n and the next one over the number of denominations available, i.e., k. Hence, the complexity of the nested loops are O(nk). The overall time complexity of the suggested algorithm is O(nk + n), which is O(nk). We have achieved a polynomial time complexity. Both arrays $minTable$ and $lastCoin$ are of size n + 1, hence the size complexity is of O(n).

```
Function minCoins(denominations d, sum n):
    initialise minTable [n+1], lastCoin [n+1]
    minTable [0]:= 0
    forall i ∈ [1, n] do
        minTable [i]:= infinite
    end
    forall i ∈ [1, n] do
        forall coins ∈ d do
            if value(coin) ≤ i then
                ans ← minTable[i − value(coin)]
                if ans ≠ infinite and minTable [i] > ans + 1 then
                    minTable [i] ← ans + 1, lastCoin [i] ← coin
                end
            end
        end
    end
    if minTable [n] = infinite then return Not possible
    sum := n  S := ϕ
    while sum ≠ 0 do
        S.insert(lastCoin [sum])
        sum ← sum - lastCoin [sum]
    end
    return S
```