Aman Gupta
2019CS10673
Arpit Chauhan
2019CS10332

Assignment - 2
COL351 - Analysis and Design of Algorithms
Term: Fall 2021
Due Date: 26^{th} September, 2021

Problem 1
Alice, Bob, and Charlie have decided to solve all exercises of the Algorithms Design book by Jon Kleinberg , EvaTardos .There are a total of n chapters, $[1, ..., n]$, and for $i \in [1, n]$, x_i denotes the number of exercises in chapter i. It is given that the maximum number of questions in each chapter is bounded by the number of chapters in the book. Your task is to distribute the chapters among Alice, Bob, and Charlie so that each of them gets to solve nearly an equal number of questions. Device a polynomial time algorithm to partition $[1, ..., n]$ into three sets S_1, S_2, S_3 so that $max\{\sum_{i \in S_1} x_i, \sum_{i \in S_2} x_i, \sum_{i \in S_3} x_i\}$ is minimized.

Solution: We will use 3-dimensional Dynamic Programming to solve this problem. Let DP[i][w_1][w_2] be true if we can obtain a three-way partition of first i chapters such that S_1 and S_2 have subset sums w_1 and w_2 respectively, otherwise false. S_3 will have subset sum, w_3 = Total - w_1 - w_2

Algorithm 1: To partition set S into three sets such that maximum subset sum is minimized

```
Initialize DP[0][0][0] = True and all other values to be False
For i = 1 to n
    For j = 0 to (i-1)*n
        For k = 0 to (i-1)*n
            DP[i][j + x_i][k] = DP[i][j + x_i][k] or DP[i − 1][j][k]
            DP[i][j][k + x_i] = DP[i][j][k + x_i] or DP[i − 1][j][k]
            DP[i][j][k] = DP[i][j][k] or DP[i − 1][j][k]
Total = 0
For i =1 to n
    Total = Total + x_i
min_so_far = n^2
For j = 0 to Total
    For k = 0 to Total
        If DP[n][j][k] = True then
            max_sum = max(j, k, Total - j - k)
            If max_sum < min_so_far
                min_so_far = max_sum
                w_1 = j
                w_2 = k

Function Partition(i,j,k,S_1,S_2,S_3){ //BACKTRACKING
    If i = 0 then
        Return
    If DP[i-1][j-x_i][k] = True then
        S_1.append(i)
        Partition(i-1,j-x_i,k,S_1,S_2,S_3)
    Else If DP[i-1][j][k-x_i] = True then
        S_2.append(i)
        Partition(i-1,j,k-x_i,S_1,S_2,S_3)
    Else
        S_3.append(i)
        Partition(i-1,j,k,S_1,S_2,S_3)
}
S_1,S_2,S_3 = φ
Backtrack(n,w_1,w_2,S_1,S_2,S_3)
return S_1,S_2,S_3
```

Runtime Analysis

It is given that the number of questions in each chapter is bounded by the number of chapters in the book i.e. $x_i \leq n$ for all i. Hence, the total number of questions across all chapters is bounded by n^2.
In lines 2-7, i takes n values, j takes upto O(n^2) values and k also ranges upto O(n^2) values and then, O(1) updates are performed on 3-D array DP, thus the time complexity for this entire calculation of DP will be O(n^5).
Then we calculate the minimum maximum subset sum by looping over all possible values of subset sums for S_1 and S_2 which are both bounded by O(n^2).Hence, this calculation takes O(n^4) time.
Next, we use backtracking to obtain the sets S_1,S_2 and S_3.In each recursive call, we append the i^{th} chapter to some set, hence time taken will be O(n).
Thus, the overall complexity comes out to be O(n^5).

Proof of Correctness

Claim 1: DP[i][w_1][w_2] is true if and only if there exists a 3-way partition of first i chapters such that S_1 and S_2 have subset sums w_1 and w_2 respectively, otherwise false.
Proof: We will use induction to prove this claim.The claim is our hypothesis with i as the hypothesis variable. Base case corresponds to $i = 0$ for which all subset sums are 0 and hence, only DP[0][0][0] is true. Let us assume that the hypothesis holds true for the first $i - 1$ chapters. Now, the i^{th} chapter with x_i questions can be present in either of the 3 subsets and hence, we have the following updates, corresponding to i^{th} chapter being assigned to S_1,S_2 or S_3 respectively.

$$DP[i][j + x_i][k] = DP[i][j + x_i][k] \; or \; DP[i - 1][j][k]$$

$$DP[i][j][k + x_i] = DP[i][j][k + x_i] \; or \; DP[i - 1][j][k]$$

$$DP[i][j][k] = DP[i][j][k] \; or \; DP[i - 1][j][k]$$

Since, the DP values upto $(i - 1)^{th}$ chapter were correct, the values obtained upto i^{th} chapter will also be correct.Hence, the claim stands proved by Induction.

Claim 1 suggests that DP[n][w_1][w_2] will be True if and only if there exists a 3-way partition of all n chapters such that S_1 and S_2 have subset sums w_1 and w_2 respectively. So, we loop over all possible values of subset sums w_1 and w_2 for which DP[n][w_1][w_2] is True and find the minimum of $max\{\sum_{i \in S_1} x_i, \sum_{i \in S_2} x_i, \sum_{i \in S_3} x_i\}$. Hence, the algorithm is correct.

To obtain the partition corresponding to optimal subset sums obtained by the algorithm, we use backtracking.
Claim 2: At every recursive step in the backtracking algorithm, the i^{th} chapter gets optimally assigned.
Proof: Using claim 1, we can say that if DP[i-1][j-x_i][k] is true, there exists a 3-way partition of first i-1 chapters such that S_1 and S_2 have subset sums j-x_i and k respectively, and hence chapter i can be added to S_1. Making similar argument for S_2 and S_3, we can say that all the chapters get optimally assigned. Also, since at every recursive step, $DP[i][j][k]$ is true, Claim 1 suggests that we will definitely find a solution. Hence, Claim 2 is correct.

Using Claim 2, we can say that the backtracking algorithm to find the optimal partition into three subsets is correct.

Problem 2.1
You are given a set C of n courses that needs to be credited to complete graduation in CSE from IITD. Further, for each c ∈ C, you are given a set P(c) of prerequisite courses that must be completed before taking the course c.
Device the most efficient algorithm to find out an order for taking the courses so that a student is able to take all the n courses with the prerequisite criteria being satisfied, if such an order exists. What is the time complexity of your algorithm? [5 marks]

Solution:
We will formulate the given problem as a graph problem, where each course denotes a vertex of the graph. The prerequisite relation will be used to make the edges of the graph. These edges will be directed because in an edge only one of the courses is a prerequisite of another and not the other way round. Hence, the set P(c) for a vertex(course) c denotes the set of vertices from which there exists an in-edge for the vertex c.

Approach
We need to find an order of the courses such that before starting a particular course all its prerequisites are covered. In terms of graphs, this means that a vertex should be present after its ancestors in order. We should note that the start of the order should be from vertices that do not have any in-edges. Hence we can place all the vertices with zero in-degree at the beginning of the order. We will further reduce the graph by removing these vertices from the graph. This will now lead to some other vertices with in-degree zero. We will now place these new vertices in order and remove them from the graph. This process will continue till the end when we obtain our complete order.

Algorithm 2: Ordering courses so that prerequisite criteria is satisfied. **Kahn's Algorithm**

```
1  Maintain a map H for the vertices and their in-degrees. Initially, this will just be the length of P(c).
2  Obtain an adjacency list representation of the problem. This list consists of out edges from each of
     the vertices.
3  Declare a queue Q.
4  Initialize the queue with the vertices of in-degree 0.
5  Maintain an output array O, that will contain the order of vertices
6  While not Q.empty() do
7      Remove an element from Q.
8      Add it to the array O
9      Reduce the in-degrees of vertices in the adjacency list of the removed element
10     If the in-degree of any of these neighboring vertices reduce to 0 then add it to Q.
11 Return O
```

Time and Space Complexity
Calculating in-degree, and making an adjacency list takes O(|V|+|E|) time. For the while loop, each of the vertices passes through the queue only once, which accounts for O(|V|), and each of the edges is used to reduces the in-degree only once, which is O(|E|). Summing up the while loop takes O(|E|+|V|) time to run.
Hence the overall complexity of the algorithm is O(|V|+|E|).
|V| = n and |E| = O(n^2) in the worst case, therefore the overall worse case complexity is O(n^2)

Additional memory is required just for the queue, the output array, and the adjacency list. The queue and the output array take O(|V|) and the adjacency list takes O(|V|+|E|) space. Therefore, the overall algorithm takes O(|V|+|E|) = O(n^2)

Proof of Correctness
We will prove the algorithm by the following assertion and claims:

Assertion The initial graph made by the courses must be a DAG for such an ordering to exists.
Proof Let us assume that there exists a cycle in the initial graph made by the courses. Take an edge (x,y) on this cycle. This states that x is a prerequisite for y so x will occur before y in the order. Take the path from y to x along this cycle, the existence of this cycle indicates that y is a prerequisite for x so y should occur before x in the order. This leads to a contradiction.

Claim A finite directed graph with no cycles always contains a vertex with in-degree zero.
Proof Let us assume that there is no such vertex that has in-degree zero. We can take any vertex of the graph. We will keep going on any one of the parents of this vertex. We cannot find any same vertex in this process because the graph does not contain a cycle. The graph only contains |V| distinct vertices therefore, the process can continue only that many numbers times. But it cannot terminate as there are no vertices with in-degree zero. This leads to a contradiction.

Claim After each iteration, the order of the output will satisfy the prerequisite criteria.
Proof We will prove the claim by induction. Let us assume that the claim is true for all iterations from 1st, 2nd ..... Up to $(i-1)^{th}$ iteration. Then the element removed in the i^{th} iteration will have all its prerequisites already satisfied. Since this element is removed from the queue, it must have in-degree zero (if not, it would have not been added to the queue) Therefore from the course removed in this iteration does not have any dependency in the remaining graph. So all its dependencies are satisfied by elements already present in the output array. So the inductive hypothesis is true and hence the claim is true.

Proof of termination
Since at least one element is removed in each iteration. The algorithm is bound to terminate in some finite time. Also, all the vertices will pass through the queue as according to one of the claims above a graph always has a vertex with in-degree zero.

Hence by the invariant and termination, we can conclude that the algorithm provides an order of courses in which the prerequisite criteria are satisfied.

Problem 2.2
Device the most efficient algorithm to find minimum number of semesters needed to complete all n courses. What is the time complexity of your algorithm? [5 marks]

Solution:
This problem requires us to find the minimum number of semesters needed to complete all n courses. Since it is an optimization problem that can be reduced into further non-overlapping subproblems we will use a dynamic programming strategy to find the minimum number of semesters.

Approach We cannot process courses that are prerequisites for one another in the same semester. As a graph problem, we are required to find the maximum depth (prerequisite sequence) of the graph. We will compute this in a recursive way with memorization. We define DP configurations as the following:
v = Vertices of the graph (represents Chapters)
dp[v] = Minimum number of semesters required to cover course v, equivalently the longest prerequisite sequence that ends on v
Initialization of the DP is done as follows,
dp[v] = 1 for all the vertices with in-degree zero
We can connect the problem into smaller sub-problems with the help of this recurrence relation,

$$\mathbf{dp[v] = 1 + \max(dp[u]) \text{ for all } u \text{ in } P(v)}$$

The recurrence is correct because to obtain the longest sequence that terminates at v, we check the longest sequence among all the neighbors(those vertices from which there is an in-edge to v) of v and add one to it, to obtain the one that terminates on u.

Algorithm 3: Finding the minimum number of semesters required to cover all n courses

```
Declare an array dp of size n.
Find all the vertices with in-degree 0, and initialize dp[v] = 0 for them.
For the rest of the vertices keep dp[v] = -1
Define a function as follows:
def eval(Vertex v):
    For u in P(v) do
        If dp[u] == -1 then
            eval(Vertex u)
        dp[v] = max(dp[v], 1+ dp[u])
        return
for v in V do
    if dp[v] == -1:
        eval(Vertex v)
return maximum(dp)
```

Time and Space Complexity
Due to memorization, the evaluation function of each of the vertex is called at most one O(|V|), and during that execution, all edges are covered only once O(|E|). Therefore the overall complexity of the algorithm is O(|V| + |E|) = O(n^2)

Space is required only for storing the dp array and therefore the memory required for this function is O(|V|) = O(n). Provided the arrays P(c) are given as input to the functions.

Proof of Correctness
We will prove the algorithm with the help of the following proofs:

Claim The number of vertices in the longest prerequisite sequence is the minimum number of semesters required to complete all n courses.
Proof This claim is true because we can arrange vertices of any other sequence (say Q) parallelly with the longest prerequisite sequence (say L). There can be two possible cases, that are:
Case-1: Q and L are disjoint
In this case, since the prerequisite sequences are independent of each other, the vertices of Q will match one by one starting from the lowest level of sequence L. Because L is the longest sequence so all vertices of Q will be terminated before L ends.
Case-2: Q and L have some common vertices
Consider the common vertex at the highest level, for the levels below this vertex the vertices of Q and L will be the same or siblings of each other. Above this vertex, L will have a larger number of vertices than Q because it is longer than Q. So Q will terminate before L.
Using the two cases above we can claim that there can be a solution that contains as many semesters as the length of the longest prerequisite subsequence Q. Therefore opt(n) <= Number(vertices(Q)). And since Q is a prerequisite sequence all its vertices need to be covered in different semesters. So opt(n) >= Number(vertices(Q))
Therefore opt(n) = Number(vertices(Longest prerequisite sequence))
Hence the claim is proved.

Claim After every stage, dp[v] stores the number of vertices in the longest chain terminating at v.
Proof The claim can be proved by structural induction. For the base case, we know that the statement is true because they have in-degree one so the number of vertices is one.
Let us assume that the induction hypothesis is true for all the neighbors of v (in P(c)), so according to the recurrence relation,
dp[v] = 1+ max(dp[u]) for all u in P(v)
The length of the longest sequence terminating at v will be one larger than the maximum terminating at one of its neighbors. The recurrence relation chooses the max of such paths from all neighbors of v. Therefore, the induction hypothesis is true for v.
Therefore the claim is true.

Using the above claims we can show that since the algorithm return the maximum element of dp array, it must be the number of vertices in the longest chain in the graph, which in turn equals the minimum number of semesters required.

Problem 2.3
Suppose for a course c $\in$ C, L(c) denotes the list of all the courses that must be completed before crediting c. Design an $O(n^3)$ time algorithm to compute the set P $\subseteq$ C×C of all those pairs (c,c_o) for which the intersection L(c) $\cap$ L(c_o) is empty. [5 marks]

Solution:
The problem asks to find all the pairs of vertices in graph G that do not have a common ancestor vertex. The list of ancestors of a node c is represented by L(c).

Approach
We can find ancestors of any vertex and then compare them for any two vertices to see if they have a common vertex or not. The ancestors can be found for any vertex by performing a DFS search on the reverse graph of G (All edges in G are reversed in direction). Once the list L is found for all vertices we can compare them of every pair of vertices to check if there is a common ancestor.

Algorithm 4: Finding all pairs that do not have a common ancestor.

```
Reverse all the edges of the graph to obtain a new graph. Perform DFS from each of the vertexes
  to find the ancestors set for the vertices.
Maintain an outset O containing all pairs of vertices satisfying the given condition.
Maintain a bit array B, of size n and initialize it will all values -1
for v in vertex set V do
    For u in vertex set V do
        if v == u then
            pass
        for x in L_u + L_v do
            B[x] = (B[x] + 1)mod 2
        If 1 is not present in B[x] and u,v are not ancestor-descendant then
            Add (u,v) to O
        Set B to all -1 values again
return O
```

Time and Space Complexity
The time required for the following processing is as follow:
Reversing the graph: O(|V|+|E|)
Performing DFS on each node of the graph: O(|V|*(|V|+|E|)
Check all pairs of points and then trying to find if they share an element or not - O($|V|^2$ * |V|)
Hence the overall time complexity is O($|V|^3$ + |V|*(|V|+|E|))
|V| = n and |E| = O(n^2)
Hence the overall complexity of the algorithm comes out to be O(n^3)

The space is required for
The revered graph - O(|V|+|E|)
L arrays of all the vertices - O(|V|*|V|)
Bit array for checking common ancestors = O(|V|) (being reused)
Therefore, the overall space complexity is O($|V|^2$ + |V| + |E|)
which is, O(n^2)

Proof of Correctness
We will prove the algorithm using the following claims:

Claim A vertex x appears before vertex y in all topological sorts if and only if they have ancestor descendant relationship.
Proof If x and y have ancestor descendant relationship then the ancestor must come before the descendant in all the topological sorts because its a prerequisite.
Now we prove the other side of the claim. We will prove this using contradiction. Let x appear before y in all the topological sorts but it is not an ancestor of y. Consider a case when we first obtain the complete ancestral graph of vertex of y (Y). Since x is not an ancestor of y then it will not occur in this graph. We now remove this ancestral graph of y from the original graph we obtain X. The vertex x is contained in X. Now consider a topological sort that first contains the topological sort of Y and then contains topological sort of X. This is a valid topological sort for the complete graph as well because there can't be vertices in X that are ancestors of vertices in Y because else they would have been ancestors of y. And in this topological sort y comes before x. So by contradiction, we can prove this part of claim.
Hence, the claim is true.

Claim Performing a DFS on the inverse graph from a particular vertex provides all the ancestors of that vertex in the original graph.
Proof Consider a vertex x, which was one of the ancestors of vertex y in the original graph. So there exists a path from x to y in the original graph. After reversing the graphs since all edges have flipped their direction, we can follow the same path as before to reach from y to x. Therefore, any arbitrary ancestor of y will be discovered in DGF from y in the inverted graph.
Suppose that there exists a vertex z in the DFS of y which was not an original ancestor of y. We can use the same arguments as before and claim that if there was such an edge z, there would have been a path from z to y in the original graph. But z is not an ancestor of y so this leads to a contradiction.
Therefore,
Ancestors of a vertex = Vertices visited by DFS in the inverted graph of that vertex.

Claim The bit array will have an element 1 if any one of the vertexes is repeated.
Proof The list of all the ancestors of a particular node is unique because in DFS traversal a visited array is present which ensures that no vertex is visited twice. So at most any element can come once in the ancestor's list L. If there is a common ancestor then it will occur exactly once in both the ancestor's lists. Hence, the value of the bit array for that vertex will become 1 (by two increments).

Using the above claims we can say that the graphs find all the vertex pairs that do not have any common ancestor. Hence, the correctness of the algorithm is proved.

Alternative Approach
Since we need to find all the vertices which do not have a common ancestor. We will first reverse each edge of the graph and make a new Graph. Now we claim that if for any two vertices (x,y) to have a common ancestor there must exist a vertex(z) in this new graph, which is at a finite distance from both x and y. In this case z will be one of the common ancestors of x and y. If no such vertex is present then x and y have no common ancestors. We can use this claim to find all such pairs. First, we apply Floyd Warshall algorithm to get distance between each pair of vertices. Then we take any two vertex x and y and check the columns of these two vertex to find a vertex that is at a finite distance from both of them. If no such vertex is found we add (x,y) to the outputs. The run time complexity of this solution is also $O(n^3)$.

Problem 3.1
Suppose you are a trader aiming to make money by taking advantage of price differences between different currencies. You model the currency exchange rates 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 (i, j), the weight of edge (i, j), say R(i,j), corresponds to total units of currency c_j received on selling 1 unit of currency c_i.
Design an algorithm to verify whether or not there exists a cycle $(c_{i_1}, ..., c_{i_k}, c_{i_{k+1}} = c_{i_1})$ such that exchanging money over this cycle results in positive gain, or equivalently, the product $R[i_1, i_2].R[i_2, i_3]...R[i_{k-1}, i_k].R[i_k, i_1]$ is larger than 1.

Solution: Let us model the given problem as a directed graph G with n nodes corresponding to n currencies $c_1, ..., c_n$ and edge weights corresponding to negative logarithm of the exchange rates between currencies i.e. $e_{ij} = -\log R[i, j]$.
We need to verify whether there exists a cycle $(c_{i_1}, ..., c_{i_k}, c_{i_{k+1}} = c_{i_1})$ such that exchanging money over this cycle results in positive gain i.e.

$$R[i_1, i_2].R[i_2, i_3]...R[i_{k-1}, i_k].R[i_k, i_1] > 1$$

Taking negative logarithm, we get

$$-\log R[i_1, i_2] - \log R[i_2, i_3].... - \log R[i_{k-1}, i_k] - \log R[i_k, i_1] < 0$$

Hence, the problem boils down to checking whether the graph G contains a negative cycle or not.

Algorithm 5: To check whether Graph G contains a negative cycle

```
1  Consider any vertex v as the source vertex.
2  Let D be the distance vector from v. Initialize D[v] = 0 and all other distances to be infinity(INF).
3  For i = 1 to n-1
4      For edge (i,j) in Edges
5          If D[i] < INF then
6              D[j] = min(D[j], D[i] + e_ij)
7  For edge (i,j) in Edges
8      If D[i] + e_ij < D[j] then
9          Return True
10 Return False
```

The above algorithm returns True if and only if the graph contains a negative cycle, otherwise returns false. In case the graph is not strongly connected, we will need to run the above algorithm on every strongly connected component.

Runtime Analysis
The initialization of the distance vector takes O(n) time. Then we perform n iterations and in each iteration we loop over all the m edges in the graph and perform the update, if required. Each update takes O(1) time and hence, the overall time complexity comes out to be O(nm). Since m is bound by $O(n^2)$, the worst case time complexity will be $O(n^3)$.

Proof of Correctness
This algorithm is called **Bellman Ford Algorithm** whose detailed analysis is done in the lectures.
Claim 1: After n-1 iterations of the for loop in lines 2-6 of the algorithm above,the array D contains the weights of shortest paths from source s to all reachable vertices, if the Graph G contains no negative weight cycles reachable from s.
Proof:Let the shortest path from source s to any reachable vertex v be $sv_2v_3...v_{k-1}v$. Since, there are no negative weight cycles in the graph, any shortest path will be a simple path with at most n-1 edges, hence $k - 1 \leq n - 1$. In i^{th} iteration, edge v_iv_{i+1} is relaxed and path distance upto v_{i+1} becomes correct(equal to the shortest path distance from s). Hence, in k-1 iterations, we will obtain the shortest path distance to v. Since $k - 1 \leq n - 1$, in n-1 iterations, the array D will contain the weights of shortest path from source s to all reachable vertices.

Claim 2: If the graph G contains a negative weight cycle, the above algorithm returns True,otherwise False

Proof: We will prove this by contradiction. Let us say that there exists a negative weight cycle $v_0v_1..v_{k-1}v_k$ in our graph i.e.

$$\sum_{i=0}^{k-1} e_{i,i+1} < 0 \tag{1}$$

Now, let us assume that the algorithm returns False in this case. Thus, D$[v_{i+1}] \leq$ D$[v_i]$ + $e_{i,i+1}$ holds for all $i = 0, 1.., k-1$. Summing over the inequality over all i, we get

$$\sum_{i=0}^{k-1} D[v_{i+1}] \leq \sum_{0}^{k-1} D[v_i] + \sum_{0}^{k-1} e_{i,i+1} \tag{2}$$

Since $v_0v_1..v_{k-1}v_k$ is a cycle, $v_0 = v_k$ and hence,

$$\sum_{i=0}^{k-1} D[v_{i+1}] = \sum_{0}^{k-1} D[v_i] \tag{3}$$

Using (3) in (2), we get $\sum_{i=0}^{k-1} e_{i,i+1} \geq 0$ which is a contradiction to (1). Hence, the claim is proved by contradiction and the **correctness of the algorithm directly follows from this claim**.
Claim 1 can also be used to justify the correctness. Since all shortest paths from source s are obtained after n-1 iterations itself, the if condition in line 8 of the algorithm never returns True if the graph does not contain a negative cycle and thus, returns True if and only if the graph contains a negative cycle,otherwise False.
Hence, the algorithm is correct.

Problem 3.2
Present a cubic time algorithm to print out such a cyclic sequence if it exists.

Solution: If we have verified that the given graph G contains a negative cycle, then we can use the algorithm given below to print out such a cyclic sequence in $O(n^3)$ time.

Algorithm 6: To print negative cycle in a Graph G given that it exists

```
1  Consider any vertex s as the source vertex.
2  Let D be the distance vector from s. Initialize D[s] = 0 and all other distances to be infinity(INF).
3  Let P be the parent vector. Initialize P[v] = -1 for all the vertices.
4  For i = 1 to n
5        temp = -1
6        For edge (i,j) in Edges
7            If D[i] < INF and D[i] + e_ij < D[j] then
8                D[j] = D[i] + e_ij
9                P[j] = i
10               temp = j
11 Cycle = φ
12 For i = 1 to n
13       temp = P[temp]
14 Cycle.append(temp)
15 v = P[temp]
16 While v != temp do
17       Cycle.append(v)
18       v = P[v]
19 Reverse the list Cycle
20 Print Cycle
```

Runtime Analysis

The initial 10 lines of the algorithm are similar to what we did in Problem 3.1 with time complexity O(nm) in general and $O(n^3)$ in the worst case. Next, we take a vertex whose distance changed in the last iteration and use the parent array to reach the cycle.This step takes O(n) time. Since, the size of the cycle is bound by the number of nodes n, further steps of building the list, reversing the list and printing the cycle take O(n) time each. Thus, the overall time complexity comes out to be $O(n^3)$ in the worst case.

Proof of Correctness

Claim 1: If the distance to some vertex gets updated in the n^{th} iteration, it must be reachable from some negative cycle.
Proof:We can prove this by contradiction. Let us suppose that there exists a vertex whose distance gets updated in the n^{th} iteration but is not reachable from any negative cycle. Claim 1 from Problem 3.1 suggests that such a vertex will have a shortest simple path of less than n edges from the source and thus gets its correct shortest path by $(n-1)^{th}$ iteration and thus no update happen in the n^{th} iteration which is a contradiction to our assumption.Thus, the claim is correct.

We are already given that such a cyclic sequence exists, i.e. the graph contains a negative cycle. Using Claim 1, we can say that in the n^{th} iteration, if the distance to some vertex is updated,that vertex must be reachable from the negative cycle. If we have a vertex whose shortest distance was updated in the n^{th} iteration, we can reach the negative cycle using it's ancestors from the parent array. Since the negative cycle will be atmost n distance away, we will reach the cycle using lines 12-13. Once we get one member of the negative cycle, we can obtain the entire cycle using the Parent array recursively.But this cycle will be in reverse order, so we reverse the list obtained before printing it.

Problem 4.1
You are given a set of k denominations, d[1], . . . , d[k]. Example for Rs. 1, 2, 5, 10, 20, 50, 100, you have d(1) = 1, d(2) = 2, d(3)=5, d(4) = 10, d(5)=20, d(6)=50, and d(7)=100.
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]. [7 marks]

Solution:
The problem of finding the maximum number of ways of making an amount from some coin denominations can be divided into overlapping sub-problems of lesser amount. Hence, we will use a **Dynamic programming** (bottom-up) to solve the problem. The constraints given in the question are amount n, and the array of coin denominations d {d[1],d[2]....d[k]} of size k.

Approach:
The solution can be formulated as 2D dynamic program which covers the dimensions of a) The index of coins array up to which we have used and b) the amount created so far. The values of the table will denote the number of possible ways to achieve that particular configuration.
Let dp denotes the table of computation, its size according to the problem is k×n+1. We define,
i → index of coins array up to which we have considered till now, lies in the range [0,k)
j → amount created so far, lies in the range [0,n]
dp[i][j] → number of ways to achieve the state with amount = j by using coins in d[:i+1]
The initialisation will be,
dp[0][i] = 0 ∀ i ∈ [1,n], we cannot make any amount using zero coins.
dp[i][0] = 1 ∀ i ∈ [0,k-1], there is only one way to make amount 0, which is empty set.
As in any DP problem, this too can be solved with a recursion on the dp table,

$$\textbf{dp[i][j] = dp[i-1][j] + dp[i][j-d[i]]}$$

The intuition behind the recurrence relation is that the state (i,j) can be filled in two mutually exclusive and exhaustive ways, either complete the amount j without using coin on i index (dp[i-1][j]) or else using the coin once (dp[i][j-d[i]). We have to return the answer as dp[k-1][n]

Algorithm 7: Number of ways to get the amount n using denominations d (size k)

```
1 Declare an empty array dp of size k×n+1.
2 Initialise it so that dp[0][j] = 0 for all j in [1,n] and dp[i][0] = 1 for all i in [0,k)
3 For i in range(k):
4     For j in range(1,n+1):
5         If j-d[i] ≥ 0 then
6             dp[i][j] = dp[i − 1][j] + dp[i][j − d[i]]
7         else
8             dp[i][j] = dp[i − 1][j]
9 Return dp[k-1][n]
```

Time and Space Complexity:
The inner loop (line 4) runs a total of n times and in each iteration a constant number of operations are performed hence it takes O(n) time. The inner loop runs for k times (Outer loop - line 3) and hence the total run time complexity is **O(k*n)**. (Polynomial).
The space complexity of the solution is also **O(k*n)** because of the table dp.
Although the space can be optimised by maintain a one dimensional array (**O(n)** space)instead of table as shown in the code below:

Figure 1: Space Optimised DP solution of the problem

```python
class Solution:
    def change(self, n: int, d: List[int]) -> int:
        dp = [0 for i in range(n+1)]
        dp[0] = 1
        for i in d:
            for j in range(n+1):
                if (j-i) >=0:
                    dp[j] += dp[j-i]
        return dp[n]

```

Proof of Correctness

We will prove the correctness of the algorithm with the following claims:

Claim The value dp[i][j] will always contain the total number of ways to make amount j using coins up to i^{th} index in d.

Proof: We will prove the claim by using strong induction.

We can begin by checking the base cases, we have dp[i][0] = 1 for all i in range 0 to k, because there is only 1 way to make amount zero, that is choose an empty set. Also dp[0][j] = 0 for j in range 1 to n because we cannot make any non-zero amount with zero number of coins.

Suppose that the hypothesis is true of all i', j' such that i'+j' < i+j

Then we have to prove that its also true for the pair i,j

There can can two exhaustive and mutually exclusive cases to make the amount j using coins in d[:i+1], these are:

Case-1: We do not use the denomination d[i] in making the amount j.

The number of way in this case will be equal to the number of ways to make the amount j using coins up to $(i-1)^{th}$ index. By the argument that the hypothesis is true for all i', j' such that i'+j' < i+j, we get the required number to be equal to dp[i-1][j]

Case-2: We have used the denomination d[i] at least once.

The number of ways possible in this case are same as the number of ways required to make amount j - d[i] using coins up to denomination till d[:i+1]. By since the hypothesis is true for all i', j' such that i'+j' < i+j, we get the required number to be equal to dp[i][j-d[i]]

Using the above two we get the total number of ways to make amount j using coins up to d[:i+1] is dp[i-1][j] + dp[i][j-d[i]] which is the same value for dp[i][j] using the recurrence relation. Therefore the inductive hypothesis is true for dp[i][j] as well.

Therefore, using the claim above we say that dp[k-1][n] represents the number of ways of making amount n by any combinations of denominations array of length k. The algorithm returns exactly that so it provides the correct solution.

Problem 4.2
Device a polynomial time algorithm to find a change of Rs.n using the minimum number of coins (again you can assume you have an infinite amount of each denomination). [8 marks]

Solution:
The problem can be subdivided into overlapping sub-problems hence we will use a Dynamic Programming approach to solve it for smaller base cases and then use a bottom-up strategy to reach the optimal solution.

Approach:
We can recursively solve the problem by reducing it to a smaller problem till it becomes trivial to solve. We will memorise the solution in an array so that we don't have to do redundant combination. We have to store a denominations in a way that we can obtain the combination of coins for the optimal solution. The state of the DP varies according to the amount, we define the following:

i $\rightarrow$ amount for which we calculate the minimum number of coins required, lies in the range [0,n]
dp[i] $\rightarrow$ minimum number of coins required to make the amount i from the denominations.
L[i] $\rightarrow$ value of the last coin used to reach the amount i in minimum number of times.
We need to initialise the dp with the base case, that is,
dp[0] = 0, the number of coins required to reach amount zero is zero.
dp[i] = math.INF for all i $\in$ [1,n+1], because initially we do not have optimal solutions to make i.
L[0] = 0, there is no coin required to reach amount 0.
Recurrence relation that connects the problems into further sub problems is the following:

$$\textbf{dp[i] = 1+ min(dp[i-j])} \ \forall j \in d \ \textbf{such that, i-j} \geq \textbf{0}$$

The intuition behind the recurrence relation is that the for each amount i we check the last coin that could be used to obtain that amount, and since we are trying to find the minimum number of coins we will use the min of all the available solutions.

Algorithm 8: Making an amount of Rs. n using minimum number of coins of denominations for array d (size k)

```
1  Declare empty arrays dp and L of size n+1 each.
2  Initialise them as dp[0] = 0 and L[0] = 0
3  dp[i] = math.INF for i = 1 to i = n
4  For i in range(1,n+1) do
5          For j in k do
6              If i-j ≥ 0 and dp[i] ≥ dp[i-j]+1 then
7                  dp[i] = dp[i − j] + 1
8                  L[i] = j
9  if dp[n] == math.INF do
10         return -1
11 count = n
12 while count != 0 do
13         print(L[count])
14         count -= L[count]
15 return 0
```

Time and Space Complexity:
The inner "for" loop(line 5) runs k number of times, and in each iteration, only a constant number of computations are done. Hence, the inner loop takes over O(k) time. This inner loop executes for each iteration of the outer loop (line 4), which runs a total of n times. In total, the two nested loops take O(k*n) amount of time. To print the optimal combination, the while loop(line 12) will run for a maximum of n times. Therefore, it will take O(n) time. Hence, overall the algorithm has **O(k*n)** time complexity.
The total space required in the algorithm is O(n) for both the array dp and L. Hence, the entire space required is of the order **O(n)**

Proof of Correctness:
We will prove the correctness of the algorithm provided by the following claims:
Claim The value of array dp[i] after i^{th} iteration of outer loop stores the minimum number of coins required to make the amount i. If it is not feasible to make amount i then it stores -1.
Proof We will prove the claim by strong induction over the amounts.
The base case are true because dp[0] = 0 as there is no coin required to make the amount zero.
Let as assume that the inductive hypothesis is true for all values up to i-1. Now we need to prove it for i as well.
We can make value i using any possible denominations from the array d[j]. To consider case by case we will consider last coin to be d[j]. The minimum number of coins required in this case is equal 1 more than the number required to make amount i-d[j]. But since the inductive hypothesis is true for all the values up to i-1, the required number to make amount i is 1+dp[i-d[j]]. The minimum number of coins required to make i will be the minima of 1 + dp[i-d[j]] over all possible j. This is exactly what we obtain from the recurrence relation in the algorithm. Hence, the inductive hypothesis is true for i as well and dp[i] contains minimum number of coins required to make amount of Rs. i.

Claim The values of coin printed by the while loop in count is the optimal combination to make the amount n.
Proof The value at the index i in the array L stores the coin value that was used last to reach the amount i in minimum number of coins. This ensured by choosing according to the minimum of DP recurrence relation. So the value L[n] stores the coin used to reach value n in minimum number of coins. So the sum total before reaching n in the minimum sequence should have been n-L[n]. The last coin used to reach this value in minimum number of coins will be L[n-L[n]]. Therefore continuing this process we can obtain all the coins that are used to make amount n with least number of coins. Hence the claim is proved.

Using the above two claims we can say that the dp[n] stores the minimum number of coin required to make the amount i, and the store L helps to get all coins of this minimum sequence of coins to make amount n. This proves the correctness of the algorithm.
