COL351 Assignment 2

September 2021

Harikesh 2019CS10355
Kangale Ankit Shriram 2019CS10363

1. algorithm

Set chapters = $\{1, 2, 3, \ldots, n\}$
Set exercises = $\{x_1, x_2, x_3, x_4, \ldots\}$

Function $triplePartition(int[]exercises)$:

n = length(exercises)
sum =0
for ele in exercises do
    sum<- sum+ele
end for

dp = $[0]|n|[0]|sum|[0] * |sum| \ldots$ 3d dp array to store smaller problems
dp[i][j][k] will store 1 if there are two disjoint subsets s and s'
in 1, 2, ... i ($\sum asx_a$) = j and ($\sum asx_a$) = k. else dp[i][j][k] =0
dp[1][exercise[1]][0] = 1
dp[1][0][0] =1
for m<-2 to n do
    for x<- 1 to sum do
        for y<-1 to sum do
            dp[m][x][y] = dp[m-1][x][y]
            or (x¿exercise[m] and
            dp[m-1][x-exercise[m]][y])or (y > exercise[m] and
            dp[m-1][x][y-exercise[m]])
        end for
    end for
end for

the block will be 1 only if there exits either two same subsets without including the m^{th} element or we add the element to first subset or we add the m^{th} element to second subset.

```
        MinvalueOfMaximumset = 99999999 ... (minimum sum of exercises of the
        maximum subset)
        Xopt = -1
        Yopt = -1
        For x<-1 to sum do
            For y<-1 to sum do
                If x>=y and y>=(sum-x-y) do
                  If dp[n][x][y] =1 do
                    If MinvalueOfMaximumset > x do
                      Xopt = x
                      Yopt = y
                      MinvalueOfMaximumset = x
                    End if
                  End if
                End if
            End for
        End for

Set first = {}
Set second = {}
Set third = {}
Function getthreesets(int[][][]dp, intxopt, intyopt, intn, int[]exercises, intsum):
        If n=1 do
          If (xopt =0 and yopt =0) do
            Add exercises[1] to third
          Else if (xopt =0) do
                  Add exercises[1] to second
          Else
                Add exercises[1] to first
          Return first, second, third
        End if
```

```
        Else do
            If (xopt-exercises[n]>0 and (dp[n-1][xopt-exercises[n]][yopt] =1)
              and (xopt-exercises[n]>= yopt >= sum-(xopt – exercises[n] +yopt)
              Add exercises[n] to first
              getthreesets(dp, xopt-exercises[n] , yopt, n-1, exercises, sum)
            Else If (yopt-exercises[n]¿0 and (dp[n-1][xopt][yopt-exercise[n]] =1)
                  and (x>= yopt – exercises[n] >= sum-(xopt – exercises[n] +y
                  Add exercise[n] to second
                  Getthreesets(dp, xopt , yopt-exercises[n], n-1, exercises, sum)
            Else
                Add exercise[n] to third
                Getthreesets(dp, xopt , yopt, n-1, exercises, sum)
            End else
        End Else
End Function
```

1) Algorithms Design Book

In the problem Alice, Bob and Charlie have decided to solve all the exercises of the book. The book has n chapters and x_i denotes number of exercises in i^{th} chapter. We have to divide the chapters among the three of them almost equally i.e. we have to partition it into three sets S_1, S_2, S_3 such that $max\{\sum x_i \forall i \in S_1, \sum x_i \forall i \in S_2, \sum x_i \forall i \in S_3\}$ is minimized. Also, maximum number of questions in each chapter is bounded by number of chapters i.e. n.

In this problem we have to find what are the three subsets (which are disjoint and exhaustive) i.e. S_1, S_2, S_3. We will try to do this using Dynamic programming approach that is we will form a 3D array for storing small subproblems i.e. we will be partitioning the given problem set into three subsets and then will store 0 or 1 into the dp array if dp[m][u][v] = 1 if there exists two distinct subsets s(m ,u , v) and s'(m, u, v) in 1, 2, 3,..... m such that ($\sum x_i \forall i \in S$) = u and ($\sum x_i \forall i \in S$) = v, else dp[m][u][v] =0.

The whole above work is done in the function triplePartition. We are assuming that u>=v and v>=sum-u-v because we have to find the minimum value of maximum sum subset and for that to remove any redundancy we are always adding elements only if the above two conditions satisfy.
After filling the table we are able to find the minimum of maximum sum subset among the three. So, we will find the sum corresponding to the two disjoint subsets it has i.e xopt and yopt. To find the elements in these three subsets we are basically going to backtrack and then find whether it belongs to the 1^{st} set or 2^{nd} set or 3^{rd} set. This we will be doing in the function getthreesets.
Thus this algorithm will finally return us our first second and third subsets of the exercises.

2a algorithm

Set C = $\{C_1, C_2, \ldots, C_n\}$ ... set of all courses
$P(C_i) = (C_a, C_b, \ldots, C_k)$ ... $\forall i \in [1, n]$
Set of all prerequisite courses of a course

Function $orderingOfCourses$:

Let $P(C_i)$ is the set of prerequisite courses for the i^{th} course
Neighbour={{} ... n times} ... List of list of siz n for represnting as graph
Indegree = {} ... list of size n
OrderOfCourses = {} ... list of size n

```
For course in C do
    Forprerequisite in P(course) do
        Add course to Neighbour[prerequisite]
        Indegree[course]<- Indegree[course] +1
    End for
End for
```

We got a directed graph $G(V, E)$ where edge (a, b)
denotes a should be completed before b
queue = [] ... for storing the courses in order

```
For i<- 1 to n do
    If Indegree[i] = 0
       Add i to queue
    End if
End for
```

```
        Count = 0
        While (size(queue)¿0)
                ele = front element of queue
                Remove ele from queue
                Count<- Count+1
                Add ele to OrderOfCourses
                For x in Neighbour[ele] do
                    Indegree[x] <- Indegree[x]-1
                    If Indegree[x] =0
                       Add x to queue
                    End if
                End for
                End while

        If (Count! = n) then
           Return {} ... (empty array as not possible to do courses)
        Else            Return orderOfCourses
        ... (order in which courses to be done)
End Function orderingOfCourses
```

Time Complexity: In this algorithm we are using the kahn's algorithm to find the topological sorting of the given subjects and their prerequisites. First of all we are iterating through the courses and forming the adjacency list for all the prerequisites given. Hence the time complexity for this is $O(n^2)$.After that we are just adding nodes with Indegree 0 and then forming the order just by visiting the nodes in it along with the neighbours of the given nodes. Thus the above algorithm will give us the order (m+n). So, finally the order of the algorithm is $O(n^2)$.

Correctness: The above algorithm terminates because the two nested for loops will take a maximum of $O(n^2)$ time while after that we are having a while loop and we are running it a maximum of n times because in the queue the maximum number of elements that can be added are n. And in each iteration one element is removed from queue. So, It will also terminate after n iterations. So, the above algorithm terminates.

Now we have to show that the above algorithm gives correct order. As while forming the adjacency list we are adding the nodes with it's neighbours and are putting those nodes in the queue which have Indegree 0. This will be the first element in the ordering because this element doesn't have a predecessor element before it. i.e. it has no prerequisite. And once it is done the degree of other elements connected to it are changed and decreased by 1 because once a predecessor is done the subjects dependent on it are free from it. Hence, this way this algorithm proceeds and always print a prerequisite before the subject because the subject will have an Indegree of 0 only when the prerequisite is done. So, it gives correct ordering of courses.

Now, one case left which is to show that whether this algorithm return that ordering is possible or not. Ordering is not possible when there is a cycle which can be seen by the fact that any element is added to queue when it is free from constraints and then removed from queue. Hence after removal it is counted. Ordering is not possible when all the elements are not added to queue or if some of them have never reached a degree of 0 which is the case when there is a cycle. Hence, if there is a cycle in the graph then this algorithm will return ordering not possible. This concludes our correctness proof.

2b

Finding minimum number of semesters to complete the graduation. It is doing all the courses that is possible only if they can be arranged in the order or if there is no cycle. So, for that the above problem reduces to finding the longest path in a Directed acyclic graph (DAG) because they have to be done in different semesters.
We can find the longest length of path in a directed acyclic graph by using dynamic programming along with Depth first search using this substructure:

(dp[course] = max(dp[course] , 1+ max(dp[neighbour1]
,dp[neighbour2],. . . ,dp[neighbourn])))

Function $minSemesters(courses, list[lists]Neighbour, list[list]P(courses))$:

We have $G(V, E)$ . . . the directed graph we got previously
Neighbour = list[lists] obtained from G(V,E) , the graph represented as the adjacency list where (a,b) represents a to be done before b.
If orderingOfCourses = {} . . . no ordering possible in it from part A
  Return maxvalue . . . (as the graph is cyclic if
  previous function returns empty list)
End if
dp = 0 *$|n + 1|$ . . . form a new array of size n+1
Visited = false*$|n + 1|$ . . . form a visited array if size n+1
minSemsReqd = 0 . . . variable to store the
minimum number of sems required

For i<-1 to n do
    If visited[i] = false
      Dfs(visited, dp, neighbour , i)
    End if
End for

```
        for i<-1 to n do
            minSemsReqd = max(minSemsReqd, dp[i])
        end for
        Return minSemsReqd
End Function

Function dfs(int[]visited, int[]dp, list[lists]neighbour, intcourse):
        visited[course] = true
        for x in neighbour[course] do
            If visited[x] = false
               Dfs(visited, dp, neighbour, x)
            End if
            dp[course] = max(dp[course] , 1 + dp[x])
        end for
end Function dfs
```

2c

Time complexity: In the above algorithm to find the minimum number of semesters to be required to do all the courses, we have a for loop and a visited array and in the upper for loop if the node is not visited we are calling the dfs function and marking all reachable nodes as visited. And we are visiting every node once and store its value in the dp array. So, time complexity of the algorithm is O(n) and space complexity is also O(n).

Correctness: First of all we have to show that this algorithm terminates. As every time we visiting a node we are marking it visited and all reachable nodes from it as visited. And storing its value In the dp array we can see that we are visiting every node once and otherwise are taking value directly from the value corresponfing to dp array. Hence, this will v=conclude when all the nodes from 1 to n are visited. Now, we have to show that this algorithm returns us the minimum number of semesters required.

Proof : *by contradiction* : Assume that there exists another possible way in which the number of semesters required are less than what is calculated by our algorithm. Then we can do all the courses in that number of semesters. What our algorithm returns is the maximum length path in the directed graph assuming there is no cycle. So, for that there remains atleast one course which is not done in those less number of semesters. Hence, we must need atleast the length of longest path semesters to complete all the courses because we can not do prerequisites and the course together. Hence, the algorithm returns us the correct solution of the minimum number of semesters

```
Function findallpairs(G):
        G'(v, E') ... we will form this graph by reversing all the edges
        Dp = [false] * |n|[false] * |n| ... dp array of size n^2 with all values true
        Arr = {} ... list to store all the pair of sets with no common prerequisites
        For v in V do
            For w in V do
                If (v=w) then
                   Dp[v][w] = true
                Else do
                   Dp[v][w] = common(v,w)
            end for
        end for

        For v in V do
            For x in V do
                If dp[v][w] = false
                   Add pair (v,x) to arr
            end for
        end for
        Return arr
End Function

Function common(vertexx, vertexy, graphG') :
        Parent = [-1] * |V| ... initialize it with -1's
        Dfs( dp, graph, course, parent)
        While (parent[x] !=-1) do
              X = parent[x]
        While (parent[y] !=-1) do
              y = parent[y]
        Return (x=y)
End Function
```

```
Function dfs(int[]dpgraphG', intcourse, int[]parent):
        Dp[course] = true;
        For (int x : neighbour(course) in reversed graph) do
            If (dp[x] = false) then
               Parent[x] = course
               Dfs(dp, G', x, parent)
        End for
End Function
```

Now in the third case we have to find all the pair of courses (c,c') for which the list of prerequisites lists L(c) intersection L(c') Is empty. We need to find an $O(n^3)$ time algorithm to compute a pair set $P \subset CC$ such that for any $(c, c) \in P$, the intersection $L(c) \cap L(c)$ is empty. First we will calculate all the prerequisites of each course by using the reverse graph of the initial graph and we need to check for each pair of courses whether the course satisfies the required condition.

Correctness: The adjacency matrix of any graph can be constructed in $O(n^2)$ by using this adjacency matrix we will get all the prerequisites of each course which need to be done before it. Then for each pair of courses (after iterating twice) we need to check whether there is any common prerequisites are not and this can be done by using the pointer of parent array formed in the DFS i.e., we iterate over the parent of each node till it reaches root node and this can be done in O(n) for each course i.e., for each pair it will be done O(n) time. So, for all n^2 pairs,it will take $O(n^3)$ time.

We have made a directed graph with new edge weights and then used Floyd Warshall algorithm to detect the negative weight cycle in the graph. A negative weight cycle is one in which the total weight of the cycle comes to be negative. The node distance from itself is always zero. But there are cases when the distance can be negative. This is when the graph contains negative weight cycle.

Time Complexity: The function newGraph is used to update the existing values of the graph with new edge weights, i.e, -log($R(i, j)$) for vertices i and j. There are a total of $\frac{n(n-1)}{2}$ edges so the running time of this function is $O(n^2)$.
Now as we have discussed in class that Floyd Warshall algorithm takes $O(n^3)$ for computing the shortest distance between the vertices. In the function gainCycle, we are using Floyd Warshall algorithm for finding negative weight cycle. So the overall complexity of our algorithm is $O(n^3) + O(n^2) = O(n^3)$.

Correctness: The algorithm must terminate for correctness of the algorithm. There are two independent nested for loops. The first for loop runs from 1 to n and the inner for loop runs from 1 to n , here there inner loop is incremented by 1 in every iterations and after every n iterations the outer for loop is incremented by 1. Inside the loops we are only updating the distance vector which takes constant time,i.e $O(1)$. So the loop will terminate after n^2 iterations.
The second nested for loop contains 3 for loops. The outermost for loop runs for n iterations. Both the 2 inner for loops runs for n iterations. The innermost loop increments by 1 in every iteration. Then middle loop increments by 1 after every n iterations while the outermost loop increments by 1 after n^2 iterations. So all the loops will terminate after n^3 iterations.
A third for loop runs for n iterations for checking if there exists any negative weight cycle in graph or not.This terminates in n iterations.

2a algorithm

Set C = $\{C_1, C_2, \ldots, C_n\}$ ... set of all courses
$P(C_i) = (C_a, C_b, \ldots, C_k)$ ... $\forall i \in [1, n]$
Set of all prerequisite courses of a course

Function $orderingOfCourses$:

Let $P(C_i)$ is the set of prerequisite courses for the i^{th} course
Neighbour={{} ... n times} ... List of list of siz n for represnting as graph
Indegree = {} ... list of size n
OrderOfCourses = {} ... list of size n

```
For course in C do
    Forprerequisite in P(course) do
        Add course to Neighbour[prerequisite]
        Indegree[course]<- Indegree[course] +1
    End for
End for
```

We got a directed graph $G(V, E)$ where edge (a, b)
denotes a should be completed before b
queue = [] ... for storing the courses in order

```
For i<- 1 to n do
    If Indegree[i] = 0
       Add i to queue
    End if
End for
```

```
        Count = 0
        While (size(queue)¿0)
                ele = front element of queue
                Remove ele from queue
                Count<- Count+1
                Add ele to OrderOfCourses
                For x in Neighbour[ele] do
                    Indegree[x] <- Indegree[x]-1
                    If Indegree[x] =0
                       Add x to queue
                    End if
                End for
                End while

        If (Count! = n) then
           Return {} ... (empty array as not possible to do courses)
        Else            Return orderOfCourses
        ... (order in which courses to be done)
End Function orderingOfCourses
```

3b Algorithm

Now if the above algorithm using Floyd Warshall algorithm shows that there exists a cycle of negative weight in the graph then we have to print the cycle.

Function $printcycle(graphG'(V, E, R')$:

```
n = |V|
m = |E|
distance = INFINITE*|V| ... distance array of size n
initilaized with INFINITE
parent = -1*|V| ... distance array of size n
initialized with -1
distance[1] =0 ... we are starting relaxing all the nodes from 1
1 is assumed to be our source node
for i<-1 to V-1 do
    for edge in E = (i, j, -log(R(i, j))) of G'(V, E, R') do
        if distance[i] != INFINITE and
          distance[i] + (-log(R(i, j))) < distance[j] do
          distance[j] = distance[i] + (-log(R(i, j)))
          parent[j] = i
        end if
    end for
end for

elementInNegativeCycle = -1
for edge E = (i, j, -log(R(i, j))) of G'(V, E, R') do
    if distance[i] != INFINITE and
      distance[i] + (-log(R(i, j))) < distance[j] do
      elementInNegativeCycle = j
      break for
    end if
end for
```

```
        Cycle = {} ... list to store the vertices of the cycle
        If (C=-1) do
          Return "NO CYCLE WITH NEGATIVE WEIGHT PRESENT"
        End if
        Else do
            Vertex = parent[elementInNegativeCycle]
            While (vertex ! = elementInNegativeCycle) do
                    Add vertex to cycle
                    Vertex = parent[vertex]
            End while
            Add elementInNegativeCycle to cycle
            Return cycle
        End else
        so, if cycle is present with negative total weight we
        will get it's vertices in order in our cycle list.
End Function
```

Time Complexity: We have used Bellman Ford algorithm for finding the negative weight cycle in the graph. This algorithm was discussed in class. In the function printcycle there are nested for loops. In the first nested for loop, the outmost for loop is running for n iterations (here n is the count of vertices in the graph) and the inner for loop is running for E iterations. So the total time taken by this for loop is $O(n * E)$ i.e. $O(V * E)$. The operations inside the for loops are merely updation if the distance values of the distance array. This take constant time. The other for loop is also ran for O(E) time for checking negative cycle in the graph. So overall effective time complexity is $O(V * E)$. The edges E = $\frac{n(n-1)}{2}$. i.e E= $O(n^2)$. So overall $O(n^3)$.

Correctness: For the correctness the loops must terminate and also we should get negative weight cycle if it is present in graph. In this algorithm of Bellman Ford, We are performing the N^th iteration for finding negative weight cycle. In this, we pick a vertex from any edge which we relax in this iteration and then using this vertex and corresponding ancestor will print negative weight cycle. Here we have made a parent array for storing the parent of each vertex in the graph. We will perform n-1 iterations and in that we will relax vertex from edges say(i,j) and store its parents in the array of parent. Again do more iterations i.e, n^th and see if there can be more vertex which can be relaxed, if not then there is no negative cycle which can be printed and we return No cycle with negative weight is present. Else if there is relaxed edge in n^th iteration, take it and take all its ancestors for printing the path of negative weight cycle.
All the loops ran during this relaxation of edges run for finite time and terminates as we are only going through the edges and total edges are bounded by $O(n^2)$. This concludes our correctness proof.

4.a Algorithm

```
Function NoOfWays(denomination, k, n)
        counter[n+1];
        counter[0] = 1;
        for i from 1 to n do
            counter[i] = 0
        end for loop

        for i from 1 to n do
            for j from 0 to k − 1 do
                if denomination[j] <= i then
                  counter[i] = counter[i] + counter[i - denomination[j]]
            end for loop
        end for loop

        return counter[n]
```

4.b Algorithm

```
Function GetMinChangeCoins(denomination, k, n):
        Value = [INFINITE]*|n + 1| ... list of size n+1 filled with INFINITE's
        IndexSetofCoins = [0]* |n + 1| ... list of size n+1 filled with 0's
        Value[0] =1
        For p <- 1 to n do
        Min = INFINITE
        For i<- 1 to k do
            If denomination[i]<= p then
              If min > 1 + value[p-denomination[i]] then
                 Index = i
                 Min = 1 + value[p-denomination[i]]
                 End if
              End if
            End for
            Value[p] = min
            IndexSetofCoins[p] = index
        End for

        x = n
        ChangeCoins = {} ... set of change coins we have
        While x>0 do
                Add IndexSetofCoins[x] to ChangeCoins
                x= x – denomination [IndexSetofCoins[x]]
        end while
        if (value[n] == INFINITE) then
          return "THIS SUM CAN'T BE FORMED USING THE DENOMINATI
        end
        return changeCoins and value
end Function
```

Our above algorithm Gives us the minimum number of changed coins by value[n] and the denominations of the coins i.e the coins used in the changeCoins list. So, we can form the sum using value[n] coins and the coins used in them are in changeCoins.

4a)

Time Complexity: The algorithm contains the main function as NoOfWays which has parameter denomination of coins, k (size of denomination set), n (value of exchange). Here the first loop is used to initialize the values of counter set to 0 which runs in $O(n)$.
Then we have nested for loops running from 1 to n which take $O(n)$ and inside it we have a for loop going from 1 to k taking $O(k)$. There are updating values of counter inside these loops which take constant time. So the Overall Time Complexity of our algorithm is $O(n * k)$.

Correctness: For the correctness part, we need to show that our algorithm terminates in finite time. Now the outer for loop runs for n iterations and inner for loop will run for k iterations which is increasing in every iteration by 1 till it becomes k. So both the loops terminate in finite time after a total of $n * k$ iterations.

Proof by induction: *base case*: when n = 0 then algorithm returns 1. This is true as for getting value 0, We don't need any coin and that is 1 way.
Induction Hypothesis: for value m, counter[m] returns the Optimal NoOfWays. Let it be x, i.e, counter[m] = x
To Prove: For value m+1, algorithm return Optimal Value.
Proof: The counter array is storing the number of ways of getting optimal value for exchange i. here we are calculating it using bottom up approach. Counter array is getting filled from 0 till m+1. In this way, when we encounter a value which is less that the given exchange we will add it as it can be used to form m+1. So the counter will give all the possible way to approach the state m+1 using all denominations of coins.

4b)

Time Complexity: The function GetMinChangeCoins has an outer loop running from 1 to n where n is the sum we want to find and the inner for loop also runs on the denominations list which is of size k. We are also finally adding the coins in our coinChange list using while loop and in every go it is decreasing x by atleast one so, it has complexity $O(n)$ but because of the nested loops the overall time complexity of this algorithm is $O(n * k)$ where n is the sum and k is the number of denominations. Also, we are using new lists values, coinChange and IndexSetOfCoins in our function. So, space complexity of the algorithm is $O(n)$.

Correctness: For proving correctness we have to first show that the above algorithm terminates. The upper part of the algorithm where we are filling our values and indexSetofCoins arrays. There are two nested for loops where the outer one is running n number of times and inner one is running k times. So, the upper part is anyways going to terminate. The second part of code where we are finding our set of optimal coins. We are running a while loop and the value of x is decreasing by atleast 1 during every iteration. So, this also terminates after n iterations. Hence our overall algorithm terminates too.
We have to also show that the value , value[n] obtained is the optimal number of change coins that we can get using the sum n. To show that this is optimal break the value array in two subarrays of size t and n-t. Now, we have to show the left half of the array value of size t is an optimal way to make change for t cents using denominations and the right half is also the optimal way to change coins for the n-b cents.

Proof by Contradiction: Assume there was a better way to obtain the optimal solution to left half of the array of size t. Then the left part of solution can be replaced with the better solution and gives a valid solution for the total n coins with fewer coins than the solution. This

contradicts the optimality of the solution, Thus the optimal solution to the coin changing problem is made up from the optimal solution of the smaller subproblems. And further we know that sum 0 can be obtained by 0 way (taking no coin) and there after we are just enhancing the optimality from 1 to n in the values array. So, values array obtained at any index gives the minimum number of change coins of sum of the given index.
When the first part of the algorithm terminates then value[p] contains minimum number of coins needed to make change for sum n and IndexSetofCoins[p] contains the first coin in the optimal solution to make change for sum p.

Proof: The base case is handled by value[0] = 0 and then the recursive formula is handling the iterations. The outer for loop is running from 1 to n and we know that denomination[i] >0 , so no element in value array is accessed before it is computed. We then calculate the minimum number of ways to get a sum of p in the p^{th} iteration using the formula $min(1 + value[p - denomination[i]])$ and then value[p] is set to this. Similarly, IndexSetofCoins[p] is set to the index of coin for which this minimum value occurs. And this optimality propagates (due to above proof) upto sum of n and we get our optimal number of coins and also the index of the coins to be used.

Finally we have to show that our set coinChange provides us the optimal set of coins for making coins. This can be proved due to the fact that Our IndexSetofCoins array will contain the first index of the coin in optimal solution and this coin is added to our coinchange set in first pass and again we know that optimal solution is the optimal substructure solution than we can get the coin we will get while making change for x - denomination[IndexSetofCoins[x]]. So, adding all these in our set gives us the optimal solution of the coins to make a change of n.