COL351-A2

Pratyush Saini 2019CS10444
Prakhar Aggarwal 2019CS50441

26th September 2021

1 Algorithms Design book

Let x_i, i $\in$ {1,2,3,...,n} denote the number of problems in chapter i. Further, it is given that $\forall i \in \{1, 2, ..., n\}$, $x_i \leq n$.
We want to partition {1,2,...,n} into three sets A, B, C to minimize the maximum of sums $x_a = \sum_{i \in A} x_i$, $x_b = \sum_{i \in B} x_i$, $x_c = \sum_{i \in C} x_i$. We intend to solve this problem using dynamic programming approach.

Define for $1 \leq m \leq n$, $u, v \geq 0$ such that $u + v \leq \sum_{i=1}^{m} x_i$.

Let $T(m, u, v)$ be the indicator function which denotes that there exists a disjoint partition of elements upto m such that sum of elements in first partition is u and in other partition is v.

$T(m, u, v) = 1$ If $\sum_{i \in A} x_i = u$ and $\sum_{i \in B} x_i = v$ for $A, B \in \{1, 2, 3..., m\}$ s.t. $A \cap B = \phi$ and 0 otherwise.
The recurrence relation therefore follows:

$$T(m, u, v) = T(m-1, u, v) \text{ or } T(m-1, u-x_m, v) \text{ or } T(m-1, u, v-x_m) \quad \forall m > 1 \tag{1}$$

After computing all $T(m, u, v)$, we need to find u and v such that T(n, u, v) = 1 and $u \leq v \leq S-u-v$ with $S - u - v$ as least as possible. (Here S = $\sum_{i=1}^{i=n} x_i$)

Algorithm 1

```
1: procedure EQUAL_SUM_PARTITION(N, Array X)                ▷ X = x_1, x_2, ..., x_n
2:     Table ← {0}_{n×n²×n²}
3:     Table[1][0][0] ← 1
4:     Table[1][x_1][0] ← 1
5:     Table[1][0][x_1] ← 1
6:     for i from 2 to N do
7:         for j from 0 to i × N do
8:             for k from 0 to i × N do
9:                 if Table[i-1][j][k] is True then
10:                    Table[i][j][k] ← True
11:                else if j − x_i ≥ 0 and Table[i-1][j-x_i][k] is True then
12:                    Table[i][j][k] ← True
13:                else if k − x_i ≥ 0 and Table[i-1][j][k-x_i] is True then
14:                    Table[i][j][k] ← True
15:                end if
16:            end for
```

```
17:         end for
18:     end for
19:     Sum_Total ← SUM(X)                                          ▷ Sum of all x_i
20:     Min_Sum ← INT_MAX
21:     Min_Sum_Pair ← (-1,-1,-1)
22:     for s_1 from 0 to N^2 do
23:         for s_2 from s_1 to N^2 do
24:             s_3 = Sum_Total - s_1 - s_2                         ▷ s_1 ≤ s_2
25:             if s_3 ≥ s_2 and s_3 < Min_Sum and Table[N][s_1][s_2] = True then
26:                 Min_Sum ← s_3
27:                 Min_Sum_Pair ← (s_1, s_2, s_3)
28:             end if
29:         end for
30:     end for
31:     Partitions P_1, P_2, P_3 ← {}, {}, {}
32:     for i from N to 1 do
33:         if Table[i][s_1 − x_i][s_2] = 1 then
34:             P_1.append(x_i)
35:             s_1 ← s_1 − x_i
36:         else if Table[i][s_1][s_2 − x_i] = 1 then
37:             P_2.append(x_i)
38:             s_2 ← s_2 − x_i
39:         else
40:             P_3.append(x_i)
41:         end if
42:     end for
43:     return P_1, P_2, P_3
44: end procedure
```

1.1 Space and Time Complexity

1.1.1 Space Complexity $O(n^5)$

The auxiliary spaces we are using in our Algorithm are:

- **Table** to store Binary values, Space Required = $O(n^5)$
- **Partition Sets** P_1, P_2, P_3 to store the Partitions of chapters in book, Space Required $O(n)$ each

So, overall space complexity = $O(n^5)$

1.1.2 Time Complexity $O(n^5)$

We run three nested For loops in the Algorithm

- **Line 6 to 18**. There are three loops, the outer one running from 1 to N, and both of the inner ones running from 0 to N^2. Each comparison takes $O(1)$ time. So, time complexity for these operations is $O(n \times n^2 \times n^2) = O(n^5)$
- **Line 22 to 30**. There are two loops both of them running from 0 to N^2. Each comparison and assignment takes $O(1)$ time. So, time complexity for these operations is $O(n^2 \times n^2) = O(n^4)$
- **Line 32 to 42**. The loop runs over a total of N iterations, and each iteration consumes $O(1)$ time in comparison and assignment. So, Time complexity for these operations = $O(n)$

So, Overall time complexity is $O(n^5)$

1.2 Proof of Correctness

Loop Invariant $\forall$ u,v such that $u + v \leq \sum_{i=1}^{m} x_i$ $T(m, u, v)$ represents a True boolean value only if $\exists$ $A, B \in \{1, 2, 3, ..., m\}$ s.t. $\sum_{i \in A} x_i = u$ and $\sum_{i \in B} x_i = v$ for $A \cap B = \phi$ and False otherwise.

Initialization: m = 1

$$T(1, 0, 0) = 1, \text{ for A} = \{\phi\} \text{ and B} = \{\phi\}$$

$$T(1, x_1, 0) = 1, \text{ for A} = \{x_1\} \text{ and B} = \{\phi\}$$

We can show that the Invariant condition gets satisfied for this case

Maintenance We can proceed inductively ahead. Suppose that the invairant is True for m = k. We will now argue that under this assumption, the invariant remains true after $k + 1^{th}$ iteration of the algorithm
Suppose $T(k, u, v) = 1$, and let $A, B \in \{1, 2, 3, ..., k\}$ be the corresponding partition sets as mentioned in the loop invariant. $\sum_{i \in A} x_i = u$ and $\sum_{i \in B} x_i = v$
Now, on adding x_{k+1} to both sides of the equations, we get

$$\sum_{i \in A \cup \{x_{k+1}\}} x_i = u + x_{k+1} (= u', say) \tag{2}$$

$$\sum_{i \in B \cup \{x_{k+1}\}} x_i = v + x_{k+1} (= v', say) \tag{3}$$

Therefore, $T(k, u' - x_{k+1}, v') \implies T(k + 1, u', v')$ and $T(k, u', v' - x_{k+1}) \implies T(k + 1, u', v')$
Trivially, we can say that $T(k, u, v) \implies T(k + 1, u, v)$, by choosing the same sets A and B, without including the element x_{k+1}
We can formally write the above implications as

$$T(k + 1, u', v') = T(k, u', v') \text{ or } T(k, u' - x_{k+1}, v') \text{ or } T(k, u', v' - x_k) \quad \forall m > 1 \tag{4}$$

Thus we can argue that at the end of $k + 1^{th}$ iteration, $T(k+1, u, v)$ is True if $\exists$ $A, B \in \{1, 2, 3, ..., k+1\}$ s.t. $\sum_{i \in A} x_i = u$ and $\sum_{i \in B} x_i = v$ for $A \cap B = \phi$ and False otherwise.

Termination After the N^{th} iteration of the algorithm, we have $T(N, u, v)$ = True for $A, B \in \{1, 2, 3, ..., N\}$ s.t. $\sum_{i \in A} x_i = u$ and $\sum_{i \in B} x_i = v$
We can now choose the third partition set to be $\{1, 2, 3, ..., N\} \backslash \{A \cup B\}$ Thus, we now have all possible sum triplets we could obtain by partition the problems in N chapters into three sets. We iterate over all sum pairs and choose our optimum answer by minimizing the maximum of the largest partition sum value. This was obtained by running two nested loops and keeping track of the Optimal answer.

1.3 Obtaining the Partitions from Optimum sum pair

We maintain three empty Partition sets, P_1, P_2, P_3. We iterate in a reverse fashion from i = N to 1, and at each iteration we compare the values of optimum sum pair with $T(i, u - x_i, v)$, $T(i, u, v - x_i)$ and $T(i, u, v)$ and add the element x_i to P_1, P_2, P_3 respectively along with updating the sum values accordingly to $s_1 - x_i$ or $s_2 - x_i$.
At the end of all the iterations, we have sets P_1, P_2, P_3 such that $\sum_{i \in P_1} x_i = s_1$, $\sum_{i \in P_2} x_i = s_2$ and $\sum_{i \in P_3} x_i = s_3$, where (s_1, s_2, s_3) is the optimum sum triplet we evaluated before.

2 Course Planner

We are given a set C of courses which needs to be credited to complete graduation. Further, $\forall c \in C$, *P(c)* denotes the prerequisites of course c that must be completed before enrolling in course c
We can construct a graph $G = (V, E)$ with the following entities:

V = Set of nodes where each node represents a particular course
E = Set of directed edges in Graph G. $\forall (x, y) \in E$ we have x as the pre-requisite of y, i.e. x must be completed before taking the course y

2.1 Part a

In this part, we need to find an optimal ordering of the courses subject to the condition that the pre-requisite criteria is satisfied. Our approach to solve this problem is by constructing a topological sorting of the courses
Recall that topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

2.1.1 Approach

Our approach is based on the following fact: **A DAG always contains a vertex with in-degree 0 and out-degree 0** (covered in Lectures).
Steps involved in finding topological sort are:

- Pre-compute the Indegree of all the vertices. Indegree of a nodes v is defined as the number of edges directed towards v in the graph. Maintain a Queue for storing vertices with indegree 0.
- Iteratively, select a vertex with In-degree 0 from the queue and add this node to the Array Topo (which contains Topologically sorted nodes). Then remove this node and all the associated edges from the Graph and Update the Indegrees of the remaining vertices.
- While updating the In-degrees of the remaining vertices, if indegree of any vertex is becoming 0, add it to the queue.
- Repeat steps 2 and 3 till the queue becomes empty. If the size of Array Topo is not equal to number of nodes at the end, then report G is not a DAG.

Finally, the Array Topo contains courses sorted in order of their undertaking such that pre-requisite criteria gets satisfied.

2.1.2 Proof of Correctness

Claim: All the edges in the Topological sorting have edges from left to right.
Proof: While appending any node v to the end of Array, the node has in-degree 0 with respected to remaining nodes. This means that there is no edge having head from the remaining elements (which will be inserted to the right) to node v. Hence, all the edges will be from left to right.
Coming back to the question, we can state that while undertaking any course c, it does not have any incoming edges from remaining courses, which signifies that there is no course from Pre-requisites of c which would be taken in the future.

What if a Cycle exists in the Graph?
We can argue that if a cycle exists in the Graph, there is no possible ordering of taking courses with fulfillment of Pre-requisite criteria.

Suppose there is cycle $v_1, v_2, v_3, ..., v_k = v_1$, Then $v_2 \in P(v_1)$, $v_2 \in P(v_3)$, ... , $v_{k-1} \in P(v_k)$, which Implies $v_1 \in P(v_1)$. Creates a contraction, so possible re-ordering exists in this case

2.1.3 Algorithm

Algorithm 2

```
1:  procedure ADJACENCY_LIST(E)
2:      AdjList ← [{}, {}...{}, {}, {}]
3:      for ∀(x, y) ∈ E do
4:          AdjList[x].insert(y)
5:      end for
6:      return AdjList
7:  end procedure
8:
9:  procedure TOPOLOGICAL SORT(V, E)
10:     Topo, Queue ← {}, {}
11:     AdjList ← ADJACENCY_LIST(V, E)
12:     Indegree ← [0, 0, 0, 0, ..., 0]
13:     for ∀(x, y) ∈ E do
14:         Indegree[y] += 1
15:     end for
16:     for ∀v ∈ V do
17:         if Indegree[v] == 0 then
18:             Queue.insert(v)
19:         end if
20:     end for
21:     while Q.empty() == False do
22:         c ← Q.front()
23:         Q.pop()
24:         Topo.insert(c)                    ▷ Insert vertex with InDegree 0 to the list
25:         for ∀ v ∈ AdjList[c] do
26:             if Indegree[v] > 0 then
27:                 Indegree[v] -= 1
28:                 if Indegree[v] == 0 then
29:                     Q.push(v)
30:                 end if
31:             end if
32:         end for
33:     end while
34:     if Topo.size ≠ |V| then
35:         return False                      ▷ There exists a Cycle
36:     end if
37:     return Topo
38: end procedure
```

2.1.4 Time and Space Complexity

The Time complexity of the above algorithm if O(|V|+|E|). The outer while loop runs at most |V| times and inner for loop runs at most |E| times.
The Space complexity of our algorithm is O(|V|) to store the indegree of vertices and Queue

2.2 Part b

In this problem, we are required to find the minimum number of semesters required to complete all the courses. An efficient algorithm to acheive this is by alloting the maximum number of courses that can be done in a semester, and there after removing all the nodes and associated edges done in current semester.

The way of choosing nodes for a particular semester therefore reduces to selecting out all the nodes having zero indegree in the current state. After removing the nodes and associated edges, we update the indegree of remaining nodes and repeat the process till termination.
Note: Here, we assume that G is a DAG, which can be checked by invoking part a above.

2.2.1 Approach

Step involved in our Algorithm are:

- Pre-compute the Indegree of all the vertices. Indegree of a nodes v is defined as the number of edges directed towards v in the graph. Maintain a Queue for storing vertices with indegree 0.
- Iterate over all the nodes in the graph and add all the nodes with Indegree 0 to the queue. Remove these nodes and associated edges from G. If size of Queue is non zero after an iterations, increment the number of semesters by 1.
- Repeat the above step till all the nodes get visited. At the end, return the variable Num_Semesters

2.2.2 Proof of Correctness

Let G be the Graph at the beginning of an iteration and G' at the end of that Iteration. Let $OPT(G)$ = Minimum number of semesters needed to complete all the courses in G.

Claim:

$$OPT(G) = OPT(G') + 1 \tag{5}$$

Proof:

- By our Algorithm, G' is obtained from G by removing all the nodes with In-degree 0 and it's associated edges.
- So, we remove all the nodes which do not have any pre-requisite or all the pre-requisites are already satisfied. Thus we can undertake all these course in one Semester
- After removing all the in-degree 0 nodes and their edges, we are left with G'. The nodes of G' having in-degree 0, all have at-least one pre-requisite node from the removed ones (if it were not so, then they also should have been removed in the current iteration)
- OPT(G) $\leq$ OPT(G') + 1
  Consider adding all the removed nodes and edges back to G'. Since OPT(G) equals the minimum semesters required in G, Hence we can say that OPT(G) $\leq$ OPT($G' \cup V_r$) = OPT(G') + 1 (Using Point 2 above), where V_r equals the set of nodes with in-degree 0
- OPT(G') $\leq$ OPT(G) - 1
  Consider removing the set V_r (as defined above) and incident edges from G. Out of the remaining Graph, OPT(G') equals the minimum semesters required in G'. Therefore, we can say that OPT(G') $\leq$ OPT($G \backslash V_r$) = OPT(G) - 1 (Using Point 2 above)
- Therefore, we finally have OPT(G) = OPT(G') + 1. The algorithm continues till we have no in-degree 0 vertex left. At each iteration, no. of Semester gets incremented by 1. So, OPT(G) equals number of Iterations performed.

2.2.3 Algorithm

Algorithm 3

```
1: procedure FIND_MIN_SEMESTERS(V, E)
2:     Queue ← {}
3:     AdjList ← ADJACENCY_LIST(V, E)                    ▷ Called from Part a
4:     Indegree ← [0, 0, 0, 0, ..., 0]
5:     Num_Semesters ← 0
6:     for ∀(x, y) ∈ E do
7:         Indegree[y] += 1
8:     end for
9:     for ∀v ∈ V do
10:        if Indegree[v] == 0 then
11:            Queue.insert(v)
12:        end if
13:    end for
14:    while Q.empty() == False do
15:        Q.push(#)
16:        while Q.front() ≠ # do
17:            c ← Q.front()
18:            Q.pop()
19:            Topo.insert(c)                    ▷ Insert vertex with InDegree 0 to the list
20:            for ∀ v ∈ AdjList[c] do
21:                if Indegree[v] > 0 then
22:                    Indegree[v] -= 1
23:                    if Indegree[v] == 0 then
24:                        Q.push(v)
25:                    end if
26:                end if
27:            end for
28:        end while
29:        Num_Semesters ← Num_Semesters + 1
30:        Q.pop()
31:    end while
32:    return Num_Semesters
33: end procedure
```

2.2.4 Time and Space Complexity

The Time complexity of the above algorithm if O(|V|+|E|). The outer while loop runs at most |V| times and inner for loop runs at most |E| times.

The Space complexity of our algorithm is O(|V|) to store the in-degree of vertices and Queue

2.3 Part c

For c $\in C$, $L(C)$ denotes the set of courses which must be credited before taking c. This can be formally stated as: For some node v $\in C$, $L(v)$ denotes the set of nodes u having a path from u to v. Again, we assume that the Graph G is DAG, which can be checked by invoking part a above. We intend to solve this question by running DFS on all nodes with reversed graph Edges.
For $v \in V$, DFS(v) returns the set of nodes u for which there exists a simple from u to v in the original graph.
In this way, we can obtain all the incompatible pairs (c, c') belonging to the set P. At the end, we just iterate over all node pairs and add to the set P if they are compatible.
Note: $\forall c \in C, \{c\} \cap L(c) = \phi$

Algorithm 4

```
1:  procedure REVERSE_ADJACENCY_LIST(E)
2:      AdjList ← [{}, {}...{}, {}, {}]
3:      for ∀(x, y) ∈ E do
4:          AdjList[y].insert(x)                          ▷ Insert Edges in reverse order
5:      end for
6:      return AdjList
7:  end procedure
8:  procedure DFS(v, AdjList, Map)
9:      Stack S ← {v}
10:     Visited ← {False, False, ..., False}
11:     Visited[v] ← True
12:     while S.empty() == False do
13:         x = S.top()
14:         for u ∈ AdjList[x] do
15:             if not Visited[u] then
16:                 Visited[u] = True
17:                 S.push(u)
18:                 if size(AdjList[u]) ≠ 0 then
19:                     Map(v, u) ← True
20:                 end if
21:             end if
22:         end for
23:     end while
24: end procedure
25: procedure FIND_INCOMPATIBLE_PAIRS(V, E)
26:     AdjList ← REVERSE_ADJACENCY_LIST(E)
27:     Incompatible (int, int) : bool ← {}              ▷ Map to store Incompatible pairs
28:     for ∀c ∈ V do
29:         DFS(c, AdjList, Incompatible)
30:     end for
31:     Set P ← {}
32:     for ∀(c, c') ∈ VXV do
33:         if Incompatible(c,c') == False and Incompatible (c', c) == False then
34:             P.push({c, c'}) and P.push({c', c})
35:         end if
36:     end for
37:     return P
38: end procedure
```

2.3.1 Proof of Correctness

Claim: $\forall c \in C$, L(c) are the set of courses reachable from c after running tree traversal on c, i.e.

$$L(c) = P(c) \cup P(P(c)) \cup P(P(P(c)))... \tag{6}$$

Proof: We can show that any course c' at level i (or depth i) after running tree traversal on c belongs to $P^i(c)$, where $P^i(c) = P(P(...(P(c)))...$ i times
This can be proven inductively.
Base case: All courses visited at level 1 (P(c)) are direct pre-requisites of c (by definition).

Suppose that the above claim is true up to level i. Then for any node v which is visited at level i+1 have at least one ancestor u from level i such that (u, v) is an edge
Now, By definition of reverse edges in our graph, v belongs to pre-requisite of c. v ∈ P(u)
and u ∈ $P^i(c)$ by Induction hypothesis. Therefore v ∈ $P^{i+1}(c)$.

Therefore, all the nodes visited after returning from tree traversal of c belongs to $P^i(c)$ for some i. L(c) by definition is the set of all courses which should be credited before taking c (represented by visited nodes).
Hence, $L(c) = P(c) \cup P(P(c)) \cup P(P(P(c)))...$

Claim: Let d be the max depth achieved after running tree-traversal on some node c. Then $\forall c' \in P(c) \cup P(P(c)) \cup ...P^{d-1}(c)$, c and c' form incompatible pair

Proof: For some c' ∈ $P^i(c)$, i < d ∃ c" ∈ $P^{i+1} \cup P^{i+2}(c) \cup ...P^d(c)$ s.t. $c'' \in L(c) \cap L(c')$. Which implies that $L(c) \cap L(c') \neq \phi$. Hence, c and c' are incompatible.
Therefore, by running DFS traversal on all nodes, we can find all incompatible pairs and exclude them from our set P.

2.3.2 Space Complexity

The auxiliary space we are using are:

- AdjList for storing edges in our Graph, which takes O(m) space.
- Stack and Visited Array at each Iteration of DFS traversal, each taking O(n) space.
- Set P to store all the compatible pair of courses, taking O(n^2) time.
- Map Incompatible to store all Incompatible pairs. Takes O(m) space.

Hence, overall time complexity of our algorithm is O(2m + n + n^2), which is O(n^2)

2.3.3 Time Complexity

- Constructing AdjList takes O(m) time, since we have to visit all the edges
- DFS traversal to a single node taken O(m + n) time. Running DFS on all nodes takes overall (mn + n^2) time
- Finding Incompatible pairs required iteration over all pairs (c, c'), which takes O(m) time.

Hence, overall Time complexity is O(mn+n^2)

3 Forex Trading

Given n currencies c_1, $c2$, ..., c_n, R(i,j) corresponds to total units of currency c_j received on selling 1 unit of currency c_i.
Mathematically, We can formulate this as a Graph G, where G has n vertices c_1, c_2, ..., c_n. Every pair of vertices have a directed edge from i to j such that **wt(i,j) = R(i,j)**. We will be using an adjacency matrix **W** to represent the edge costs.

3.1 Designing an Algorithm

We need to find whether there exists a cycle $(c_{i_1}, \ldots, c_{i_k}, c_{i_{k+1}} = c_{i_1})$ such that the

$$W[i_1, i_2] * W[i_2, i_3] \ \ldots \ W[i_{k-1}, i_k] * W[i_k, i_1] > 1 \tag{7}$$

Taking log on both sides, we get

$$\begin{aligned} log(W[i_1, i_2] * W[i_2, i_3] \ \ldots \ W[i_{k-1}, i_k] * W[i_k, i_1]) > log(1) \ \ i.e., \\ => log(W[i_1, i_2] * W[i_2, i_3] \ \ldots \ W[i_{k-1}, i_k] * W[i_k, i_1]) > 0 \\ => log(W[i_1, i_2]) \ + \ log(W[i_2, i_3]) \ + \ \ldots \ + \ log(W[i_{k-1}, i_k]) \ + \ log(W[i_k, i_1]) > 0 \\ => (-log(W[i_1, i_2])) \ + \ (-log(W[i_2, i_3])) \ + \ \ldots \ + \ (-log(W[i_{k-1}, i_k])) \ + \ (-log(W[i_k, i_1])) < 0 \end{aligned}$$

So, we can construct a graph G' such that both have the same set of vertices and the adjacency matrix is given by W' where W'[i,j] = (-log(W[i,j]) i.e. we replace the edge weights with their negative logarithm
The updated condition for positive gain changes to

$$W'[i_1, i_2] \ + \ W'[i_2, i_3] \ + \ \ldots \ + \ W'[i_{k-1}, i_k] \ + \ W'[i_k, i_1] < 0 \tag{8}$$

So, the Forex Trading problem can be formulated as a problem of finding negative cycle in G'. This problem can be easily solved using Bellman-Ford algorithm (Algorithm 5).

3.2 Algorithm

Algorithm 5

```
1:  procedure DETECT_NEGATIVE_CYCLE(G, W)                    ▷ W[i,j] = R(i,j)
2:      for ∀ (u,v) ∈ G(E) do
3:          W[u,v] = -log(W[u,v])
4:      end for
5:      s ← source vertex
6:      D ← distance list of size n
7:
8:      for v from 1 to n do
9:          D[v] ← INFINITY
10:         if v = s then
11:             D[v] ← 0
12:         end if
13:     end for
14:
15:     for i from 1 to n-1 do
16:         for ∀ (u,v) ∈ G(E) do
17:             D[v] ← min(D[v], D[u] + W[u,v])          ▷ Standard Bellman-Ford algorithm
```

```
        end for
    end for

    for ∀ (u,v) ∈ G(E) do
        if D[u] + W[u,v] < D[v] then
            return True                          ▷ Graph has a negative weight cycle
        end if
    end for
    return False
end procedure
```

Computing the Cyclic Sequence: For computing the cyclic sequence, we will maintain a predecessor list of length |G(V)| which will store the previous vertex (let say u) on the shortest path from source (s) to v i.e. dist(s,v) = dist(s,u) + wt(u,v).

Algorithm 6

```
procedure NEGATIVE_CYCLE_SEQUENCE(G, W)                    ▷ W[i,j] = R(i,j)
    for ∀ (u,v) ∈ G(E) do
        W[u,v] = -log(W[u,v])
    end for
    s ← source vertex
    D ← distance list of size n
    P ← predecessor in the shortest path

    for v from 1 to n do
        D[v] ← INFINITY
        P[v] ← -1
        if v = s then
            D[v] ← 0
        end if
    end for

    for i from 1 to n-1 do
        for ∀ (u,v) ∈ G(E) do
            if D[u] + W[u,v] < D[v] then
                D[v] ← D[u] + W[u,v]
                P[v] ← u                   ▷ u is predecessor of v in the shortest path
            end if
        end for
    end for

    cyclic_seq ← []
    for ∀ (u,v) ∈ G(E) do
        if D[u] + W[u,v] < D[v] then
            temp ← u
            while temp ≠ v and temp not in cyclic_seq  do
                cyclic_seq.append(temp)
                temp ← P[temp]
```

```
            end while
            if temp not in cyclic_seq then
                cyclic_seq.append(v)
            end if
            return cyclic_seq
        end if
    end for

    return cyclic_seq                                   ▷ empty list (no negative cycle)
end procedure
```

3.3 Proof of Correctness

3.3.1 Part a

- We have shown in section 3.1 that finding a cycle for which the product of weights of the edges of the graph is greater than 1 is equivalent to finding if there exists a cycle or not for which the sum of the weights of the edges of that cycle is negative with weights W[i,j] = -log(R[i,j])
- For detecting if there is a negative cycle in a graph G, we have used Bellman-Ford algorithm (proof of correctness for Bellman-Ford is covered in lectures)

3.3.2 Part b

For computing the cyclic sequence, we have modified Bellman-Ford algorithm to have a predecessor list which stores the previous vertex on the shortest path from s to v (explained before the pseudocode of Algorithm-6)
We compute the shortest paths in n-1 iterations. After that, we iterate over all the edges (N^{th} iteration). If there is any relaxation in the N^{th} iteration, there is a negative cycle in G.

Claim: For any vertex v present in a negative Cycle, the sequence v,P[v], P[P[v]],...,v gives the negative cycle.
Proof: Assume for contradiction that the cycle is not negative. Since the vertex v which is detected when relaxation is detected on N^{th} iteration, it is sure to be a vertex in negative cycle.
We have already stored the Parent nodes of each child corresponding to **shortest distance** path from source s. For computing the cycle, we loop over the predecessor vertices using temp after we detect an edge which is relaxed. If temp is already in **cyclic_seq**, then we terminate the while loop and return the cycle.
Since, the cycle's weight is positive and also corresponds to the shortest path, it implies that any other cycle for which v is a part of has more weight that current value, which contradicts what that is a negative weight cycle containing v.

3.4 Time and Space Complexity

- **Algorithm 1 (Detecting Negative Cycle)**
  - First "for" loop is O(m) where m is the number of edges. Second "for" loop is O(n)
  - For the next loop, we iterate over all the edges n-1 times => O((n-1)*m) i.e. **O(mn)**
  - In the last "for" loop, we iterate over the edges to check if there is any updation in the shortest distance, i.e. O(m).
  - So, overall Time Complexity: O(mn). We know number of edges is O(n^2) (assuming edge between every pair of vertices), we get **Time Complexity: O(n^3)**

- We need n*n matrix for adjacency matrix representation. D is a linear list. So, overall **Space Complexity: O(n^2)**

- **Algorithm 2 (Negative Cycle Sequence)**
  - First "for" loop is O(m) where m is the number of edges. Second "for" loop is O(n)
  - For the next loop, similar to the previous algorithm, we iterate over all the edges n-1 times => O((n-1)*m) i.e. **O(mn)**
  - In the last "for" loop, we iterate over the edges to check if there is any updation in the shortest distance. If yes, then we loop over the cycle using the "Predecessor list **P**" and return the cycle.
  - Complexity of this loop will be O(m) + O(n) cause max length of cycle can be n. As m = O(n^2), complexity of this loop: O(m).
  - So, overall Time Complexity: O(mn). We know number of edges is O(n^2) (assuming edge between every pair of vertices), we get **Time Complexity: O(n^3)**
  - We need n*n matrix for adjacency matrix representation. D, P needs linear space. So, overall **Space Complexity: O(n^2)**

4 Coin Change

4.1 Part a

Given an array **d** of size k, we need to compute the number of ways to make change for Rs n given an infinite amount of coins/notes of each denomination.

4.1.1 Algorithm

Approach:

- Let count[n,i] denote the number of ways in which we can represent n using first i values of array d.
- If n = 0, then count[n,i] will be 1 (only possible way will be with no coins).
- Else, for the i^{th} coin, we either take that coin in the solution or we don't. If it is not included, then number of ways will be count[n,i-1] (representing n using i-1 coins (if i>1)). If included, then number of ways will be count[n-i,i] (if n-i $\geq$ 0) (representing n-i using i coins).
- So, if n>0, then count[n,i] = count[n,i-1] + count[n-i,i], else, for n = 0, count[n,i] = 1 $\forall$ i $\in \{1, 2, ..., k\}$

Algorithm 7

```
1:  procedure NUMBER_OF_WAYS_TO_REPRESENT_N(d, n, k)      ▷ d: denomination list of size k
2:      count ← count array of size (n+1)*k
3:      d ← denomination array of size k
4:      for i from 1 to k do
5:          count[0][i] ← 1
6:      end for

7:      for val from 1 to n do
8:          for i from 1 to k do
9:              denom ← d[i]                                ▷ denom denotes the i^th denomination
10:             denom_included ← 0
11:             denom_excluded ← 0
12:             if i > 1 then
13:                 denom_excluded ← count[val,i-1]
14:             end if
15:             if val - denom ≥ 0 then
16:                 denom_included ← count[val-denom,i]
17:             end if
18:             count[val,i] ← denom_included + denom_excluded
19:         end for
20:     end for
21:     return count[n,k]
22: end procedure
```

4.1.2 Proof of Correctness

- For computing the number of ways to make change for **val** using first i denominations, we take two mutually exclusive case for the i^{th} denomination.

- Either we include the i^{th} denomination (if possible) for making change for val and compute change for "val-d[i]" using first i denominations, or we not include the i^{th} denomination for making change for val and compute change for "val" using first i-1 denominations and take the sum for computing change of val using first i denominations.
- For the case when val = 0, number of ways will be 1 i.e. to take no coins.

4.2 Part b

Given an array **d** of size k, we need to compute the minimum number of coins to make change for Rs n given an infinite amount of coins/notes of each denomination.

4.2.1 Algorithm

Approach:

- Let minCoins[n,i] denote the minimum coins needed to represent n using first i values of array d.
- If n = 0, then minCoins[n,i] will be 0 (only possible way will be with **0** coins)
- Else, for the i^{th} coin, we either take that coin in the solution or we don't. If it is not included, then min coins needed will be minCoins[n,i-1] (representing n using i-1 coins). If included, then min coins needed will be minCoins[n-i,i] + 1 (if n-i $\geq$ 0) (representing n-i using i coins).
- So, if n>0, then minCoins[n,i] = min(minCoins[n,i], minCoins[n,i-1], minCoins[n-i,i] + 1), else, for n = 0, minCoins[n,i] = 0 $\forall$ i $\in \{1, 2, ..., k\}$

Algorithm 8

```
1:  procedure MIN_COINS_TO_REPRESENT_N(d, n, k)          ▷ d: denomination list of size k
2:      minCoins ← array of size (n+1)*k
3:      d ← denomination array of size k
4:      P ← predecessor array of size (n+1)
5:      for i from 1 to k do
6:          minCoins[0][i] ← 0
7:      end for
8:      for val from 1 to n do
9:          P[val] ← -1                                  ▷ predecessor
10:     end for

11:     for val from 1 to n do
12:         for i from 1 to k do
13:             denom ← d[i]                  ▷ denom denotes the i^th denomination
14:             denom_included ← INFINITY
15:             denom_excluded ← INFINITY
16:             if i > 1 then
17:                 denom_excluded ← minCoins[val,i-1]
18:             end if
19:             if val - denom ≥ 0 then
20:                 denom_included ← minCoins[val-denom,i] + 1
21:             end if
22:             if min(denom_included, denom_excluded) < minCoins[val, i] then
```

```
23:                 if denom_included < denom_excluded then
24:                     P[val] ← i
25:                     minCoins[val,i] = denom_included
26:                 end if
27:             end if
28:         end for
29:     end for
30:     return minCoins[n,k]
31: end procedure
```

4.2.2 Proof of Correctness

- For computing the minimum coins needed to make change for **val** using first i denominations, we take two mutually exclusive case for the i^{th} denomination.
- Either we include the i^{th} denomination (if possible) for making change for val and compute (change for "val-d[i]" using first i denominations) + 1, or we not include the i^{th} denomination for making change for val and compute change for "val" using first i-1 denominations and take the minimum of both for computing minimum coins needed to make change for **val** using first i denominations.
- For the case when val = 0, number of ways will be 0 i.e. to take no coins (zero coins).

4.2.3 Time and Space Complexity

- Part a
  - First "for" loop is O(k). For **Lines 7-20**, we have 2 "for" loops, for each value from 1 to n, inner "for" loop runs over k denominations, hence, O(n*k)
  - So, overall **Time Complexity: O(nk)**
  - We are using a count array of size (n+1)*k and a denomination array of size k. Overall **Space Complexity: O(nk)**
- Part b
  - First "for" loop is O(k) and second "for" loop is O(n). For **Lines 11-29**, we have 2 "for" loops, for each value from 1 to n, inner "for" loop runs over k denominations, hence, O(n*k)
  - So, overall **Time Complexity: O(nk)**
  - We are using a minCoins array of size (n+1)*k, a denomination array of size k and a predecessor array of size n. **Space Complexity: O(nk)**