COL351
Assignment 2

Mallika Prabhakar (2019CS50440)
Sayam Sethi (2019CS10399)

September 2021

Contents

1 Question 1

Question 1

Question. *Alice, Bob, and Charlie have decided to solve all exercises of the Algorithms Design book by Jon Kleinberg, Éva Tardos. There are a total of n chapters, $[1, \ldots, n]$, 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 $[1, \ldots, n]$ 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 propose a *Dynamic Programming* solution for this problem. The idea is to generate all possible combinations of S_1, S_2, S_3 and then find the best combination of out them. The naïve solution will have an exponential complexity ($O(3^n)$) and hence it needs to be modified so that it can be executed in polynomial time complexity.
We make the following observations to optimise our solution:

1. To find the optimal paritition of S, only the sum of each of S_1, S_2, S_3 matters
2. Order of picking elements for each set doesn't affect the solution
3. Fixing the sum of S_1 and S_2 uniquely identifies the sum of S_3

Using these observations we come up with the following DP table:

$$\begin{aligned} dp(i, s1, s2) &= dp(i-1, s1 - S[i], s2) \vee dp(i-1, s1, s2 - S[i]) \vee dp(i-1, s1, s2) \\ &\quad \forall i \in \{1, \ldots, n\}; s1, s2 \in \{1, \ldots, sum(S)\} \\ dp(i, p, q) &= \perp, \ \forall i \in \{1, \ldots, n\}; p, q < 0 \\ dp(0, 0, 0) &= \top \end{aligned} \tag{1}$$

where $dp(i, s1, s2)$ is $\top$ if we can generate atleast one partition using the first i elements such that any two partitions have sums $s1$ and $s2$

Claim 1.1. *The dp table generated using the Equation 1 is correct, i.e., $dp(i, s1, s2) = \top$ iff there exists a partition using the first i elements with sums $s1, s2, sum(S[1:i]) - (s1 + s2)$*

Proof. We will prove the correctness of the claim by induction on i.
Base case: $i = 0$
$dp(0, s1, s2)$ is $\top$ only when $s1 = s2 = 0$ and $\perp$ otherwise. We know that we can generate only three empty sets using the first 0 elements and thus their sums will all be 0. Therefore the base case is true.
Induction step: Assume that the claim is true for $i - 1$. Consider $dp(i, s1, s2)$,
The i^{th} element will be present in exactly one of S_1, S_2, S_3. Therefore, we have three cases:

1. $S[i]$ is in S_1, then the sum of S_1 upto the first $i - 1$ elements will be $s1 - S[i]$ and the sum of the other two sets doesn't change

2. $S[i]$ is in S_2, then the sum of S_2 upto the first $i-1$ elements will be $s2 - S[i]$ and the sum of the other two sets doesn't change

3. $S[i]$ is in S_3, then the sum of S_3 upto the first $i-1$ elements will be $sum(S[1:(i-1)]) - (s1 + s2)$ and the sum of the other two sets doesn't change

Thus, the only possibilities for valid partition sums using the first i elements are exactly those when we can generate atleast one of the above three paritions using the first $i-1$ elements. The transition equation given in Equation 1 exactly captures this. We have shown that for $dp(i, s1, s2)$ to be $\top$, atleast one of $dp(i-1, s1-S[i], s2), dp(i-1, s1, s2-S[i]), dp(i-1, s1, s2)$ must be $\top$. From induction, we know that the $(i-1)^{\text{th}}$ row of the table is true iff there exists a valid parition. Therefore, if $dp(i, s1, s2) = \top$ then there exists a partition using the first i elements with sums $s1, s2, sum(S[1:i]) - (s1 + s2)$. ($\implies$)

We still have to prove the converse, i.e., if there exists a partition using the first i elements with sums $s1, s2, sum(S[1:i]) - (s1 + s2)$, then $dp(i, s1, s2) = \top$.
To prove this, we observe that the dp table considers all possible sums since $s1, s2$ iterate in the range $\{1, \ldots, sum(S)\}$. Therefore, if there exists a valid solution, the dp table considers it and is assigned $\top$. Else it is assigned $\perp$. ($\Longleftarrow$) $\square$

Now that we have proved the correctness of Equation 1, we present an algorithm for computing the same:

Algorithm 1 DP solution for partitioning

```
procedure PARTITION(S)
    n ← size(S)
    s ← sum(S)
    dp ← table of size (n + 1) × (s + 1) × (s + 1) initialised with ⊥
    dp(0, 0, 0) ← ⊤
    for i in [1, n] do
        for s1 in [0, s] do
            for s2 in [0, s] do
                dp(i, s1, s2) ← dp(i − 1, s1, s2)
                if s1 ≥ S[i] then
                    dp(i, s1, s2) ← dp(i, s1, s2) ∨ dp(i − 1, s1 − S[i], s2)
                end if
                if s2 ≥ S[i] then
                    dp(i, s1, s2) ← dp(i, s1, s2) ∨ dp(i − 1, s1, s2 − S[i])
                end if
            end for
        end for
    end for
```

19: $bestPair \leftarrow (-1, -1)$
20: $leastMax \leftarrow \infty$
21: **for all** $(s1, s2) \in [0, s] \times [0, s]$ **do**
22: **if** $dp(n, s1, s2) = \top$ and $\max(s1, s2, s - (s1 + s2)) < leastMax$ **then**
23: $leastMax \leftarrow \max(s1, s2, s - (s1 + s2))$
24: $bestPair \leftarrow (s1, s2)$
25: **end if**
26: **end for**
27: **return** $Backtrack(dp, S, bestPair)$ ▷ return the partition after backtracking on the DP table
28: **end procedure**

The `Backtrack` routine used in the last step generates the partitions which give the optimal values of the sum. It is given as:

Algorithm 2 Backtracking to generate parition

procedure BACKTRACK$(dp, S, (s1, s2))$
  $n \leftarrow size(S)$
  $S_1 \leftarrow \{\}$
  $S_2 \leftarrow \{\}$
  $S_3 \leftarrow \{\}$
  for i in $[n, 1]$ **do**
    if $s1 \geq S[i]$ and $dp(i-1, s1 - S[i], s2) = \top$ **then**
      $S_1 \leftarrow add(S_1, i)$
      $s1 \leftarrow s1 - S[i]$
    else if $s2 \geq S[i]$ and $dp(i-1, s1, s2 - S[i]) = \top$ **then**
      $S_2 \leftarrow add(S_2, i)$
      $s2 \leftarrow s2 - S[i]$
    else
      $S_3 \leftarrow add(S_3, i)$
    end if
  end for
  assert $(s1 = 0$ and $s2 = 0)$
  return (S_1, S_2, S_3)
end procedure

We have shown the correctness of the DP table in Claim 1.1, therefore, to prove correctness of Algorithm 1, we need to show the correctness of the `for` loop from lines $21 - 26$ and Algorithm 2. The `for` loop iterates over all possible states of $(s1, s2)$ and finds the best state out of the valid states (where $dp(i, s1, s2) = \top$). Since the states explored by the `for` loop are exhaustive, the optimal solution is selected.
To prove the correctness of Algorithm 2, we notice that $dp(i, s1, s2)$ is $\top$ only if there exists a valid partitioning. Therefore, when updating $i, s1, s2$ in the `for` loop, we move from one valid partition to another. Therefore, Algorithm 2 generates the correct partitioning. This

completes the proof of correctness for Algorithm 1.

Space complexity: We create a *dp* table of size $(n+1)\times(s+1)\times(s+1)$ in Algorithm 1 and we generate the sets S_1, S_2, S_3 which have a total size of n in Algorithm 2. We use constant space everywhere else. Therefore the total space complexity of the solution is

$$O(n\times s\times s)+O(n)=O(n\times s^2)=O(n\times(n\times\max(S))^2)=O(n\times(n\times n)^2)=O(n^5) \quad (2)$$

Time complexity: We run nested `for` loop in Algorithm 1 (lines $6-18$) having a total of $n\times(s+1)\times(s+1)$ iterations and each iteration taking $O(1)$ time. We run another `for` loop (lines $21-26$) which has $(s+1)\times(s+1)$ iterations and each iteration again takes $O(1)$ time. We run a `for` loop of n iterations in Algorithm 2 and each iteration takes $O(1)$ time (*add* can be implemented in $O(1)$ using a list or vector). All other operations take $O(1)$. Therefore the total time complexity is given by:

$$O(n\times s\times s)+O(s\times s)+O(n)=O(n\times s^2)=O(n^5) \quad (3)$$

We have thus obtained a polynomial time algorithm for parititoning the given set S into S_1, S_2, S_3 such 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. □

2 Question 2

2.1 2.1

2.1

Question. *Given a set of C courses, devise 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?*

Solution. In this problem, an order of completing courses exists if there is no loop in the graph. A directed graph (G) is generated by setting all courses as vertices (V or C) and the pre-requirements of a course as edges (E) which is directed from the pre-requisite course to the course which needs it as pre-requisite. We use topological sort for obtaining a solution and return an empty array if no order exists (i.e., there is a loop in the graph).

Algorithm 3 The way in which courses can be completed

```
procedure ORDER(G)
    adjList ← generateAdjList(G)                      ▷ Creates an adjacency list in O(V + E) time
    colour ← new dictionary()
    colour[node] ← 0 for all nodes                     ▷ 0 represents unvisited
    tOrder ← []
    valid ← true
    for all node in G do
        if colour[node] = 0 then                       ▷ node is unvisited
            perform DFS(node, adjList, colour, valid)  ▷ directly alters variables at source
        end if
    end for
    if valid = true then
        return reverse(tOrder)                         ▷ tOrder should be reversed
    else
        return []
    end if
end procedure
```

Algorithm 4 DFS function

```
procedure DFS(node, adjList, colour, valid)
    colour[node] = 1
    for all neighbours in adjList[node] do
        if colour[neighbour] = 0 then                ▷ neighbour is also unvisited
            DFS(neighbour, adjList, colour, valid)
        else if colour[neighbour] = 1 then           ▷ Implies cycle exists
            valid ← false
            return
        end if
    end for
    color[node] ← 2                                  ▷ node is fully processed
    push node in tOrder                              ▷ add node at the end
end procedure
```

Proof of correctness. The requirement for a course is that its prerequisites must be completed before the course is taken. Let the ordered list of courses be given by T. Now, the requirement can be mathematically formulated as

$$\forall c_1, c_2 \in C : prereq(c_1, c_2) \implies pos_T(c_1) < pos_T(c_2) \tag{4}$$

Here $pos_T(c)$ gives the position of course c in the list T. We know that a course c_1 is a prereq of course c_2 iff there is an edge (c_1, c_2) in G, the graph we have formulated. Therefore, the Equation 4 is equivalent to saying that:

$$\forall c_1, c_2 \in C : (c_1, c_2) \in E(G) \implies pos_T(c_1) < pos_T(c_2) \tag{5}$$

This is exactly the condition for a topological sort on graph G. Therefore we compute the topological sort on G using the DFS algorithm discussed in class. The only difference is that instead of computing the finish times and then sorting the vertices, we make the observation that the order of increasing finish times are equivalent to the order in which DFS returns from each vertex. Therefore we push vertices in this postorder fashion and then reverse the list obtained to generate the topological sorting on G.
This completes the proof of correctness of our approach and Algorithm 3 □

Proof of termination. To check if the algorithm terminates, as for loop has finite number of steps, we need to check the termination of DFS function. The for loop in DFS terminates if-

1. There is a cycle
2. All the *neighbours* are processed

Since the number of *neighbours* are finite, Condition 2 also terminates after finite steps. Moreover, the number of times DFS function in the *Order* function is called can not be larger than $|V|$. Hence the algorithm terminates. □

Time Complexity. Following are the methods to be considered for analysind time complexity-

- Time to make adjList: $O(V+E)$ (trivial)
- Time to make colour: $O(V)$ (trivial)
- Time taken by main for loop of $Order$: $O(V+E)$ (since DFS)

Hence the overall time complexity of the $Order$ function becomes $O(V+E)$ □

Space Complexity. Space complexity is determined by size of $adjList$ adjacency list and $colour$ array. Complexity=$O(V+E)+O(V)$=$O(V+E)$ □

□

2.2 2.2

2.2

Question. *Given a set of C courses, devise the most efficient algorithm to find minimum number of semesters needed to complete all n courses. What is the time complexity of your algorithm?*

Solution. We have courses labeled 1 to n and we have to find the minimum number of semesters taken to complete the degree requirement. We first check if the graph is a DAG or not using function $Order$ in question 2.1. If a cycle exists, we can never complete degree requirement so we $return - 1$ in that scenario. Else we use a *queue* involving *indegree* of all courses and assign semester to all courses in the array *sem*. The maximum value in *sem* will be the minimum number of semesters needed to complete all the courses.

Algorithm 5 Least number of semesters for completion

```
procedure SEMESTER(G)
    existence ← Order(G)                                          ▷ from 2.1
    if existence = [] then                                        ▷ no solution case
        return -1
    end if
    adjList ← generateAdjList(G)          ▷ Creates an adjacency list in O(V+E) time
    indegree ← [0 for i in range(size(V) + 1)]
    sem ← [0 for i in range(size(V) + 1)]
    q ← empty_queue()
    for i in adjList do                           ▷ calculate indegree for all vertices
        for j in adjList[i] do
            indegree[j] ← indegree[j] + 1
        end for
    end for
    for i in range(1,size(V)+1) do    ▷ initialise sem and fill the queue for indegree=0
        if indegree[i] = 0 then
            sem[i] ← 1
            push(q, i)
        end if
    end for
    while size(q) ≠ 0 do
        node ← pop(q, 0)                                          ▷ queue is FIFO
        for neighbour in adjList[node] do
            indegree[neighbour] ← indegree[neighbour] − 1
            if indegree[neighbour] = 0 then
                sem[neighbour] ← sem[node] + 1       ▷ Pre-requisite in previous semester
                push(q, neighbour)
            end if
        end for
    end while
    return max(sem)
end procedure
```

□

Proof of correctness. **Case 1:** Graph has a cycle

If the graph of courses has a cycle, it implies that the courses can never be completed since the prerequisite of a course is dependent on the course. To check cycles in the graph, we use an O(V+E) time algorithm *Order* made in question 2.1 which return an empty array [] if graph contains a cycle since a topological order would not be possible. If the result of function $Order(G)$ is [], we return -1 denoting completion of all courses is impossible.

Case 2: Solution exists

Here, the invariant is that after every iteration of while loop, $\max(curr\_iter) \leq$

$\max(previous\_iter) + 1$, i.e., the maximum number of semesters needed to complete the courses until the courses *visited* until current index is not more than 1+ maximum number of semesters needed for courses until the previous iteration

Proof by induction

Base case: The base case is true since in the first iteration, we update $sem[node]$ by 1 which will be < 2 since all nodes in the queue had $sem[node] = 1$.

Inductive step: Assume that the claim is true for $(i-1)^{\text{th}}$ iteration, consider the i^{th} iteration. We notice that the nodes present in the queue have a value which is one of $\max(previous\_iter)$ or $\max(previous\_iter) - 1$. Therefore, if the node that is updated is added in the queue, it will have a value $\leq \max(previous\_iter) + 1$. Therefore, we have shown the inductive claim to be true for the i^{th} iteration.

This completes the proof of induction and hence the invariant of the while loop.

We now show that each course is done as early as possible. From the claim of induction, we have shown that the maximum semesters needed until the i^{th} iteration does not exceed $\max(previous\_iter) + 1$. Therefore, the new course is done in the same semester as $\max(previous\_iter)$ or in the next semester. In the case when it is done in the next semester, it could not have been done in an earlier semester since the indegree would have been > 0. Therefore we have shown that our algorithm returns the most optimal solution.

$\square$

Proof of termination. *Semester* function ends if either the course graph is not a directed acyclic graph since the courses can never be completed or after completion of algorithm. For the algorithm, the initial steps take finite amount of time and for loops are bound by V. for the *while* loop, it will run until the *queue* becomes empty. The algorithm effectively adds all *vertices* in the *queue* at different times when the *indegree* of a node turns 0 which implies there can be no duplicate additions. Since an element is popped from the queue in every iteration of while loop, while loop runs are finite and bound by V. Hence the algorithm terminates. $\square$

Time Complexity. Following are the time complexities for different parts-

- Time to make *adjList*: $O(V + E)$ (trivial)
- Time to make *indegree*: $O(V)$ (trivial)
- Time to make *sem*: $O(V)$ (trivial)
- Time taken by *while* loop: $O(V + E)$
- Time taken by *max* function: $O(V)$ (linear)

Time taken by while loop is $O(V + E)$ because for termination of algorithm, it has to go through every node and it's neighbours once. Hence the overall time complexity of the *Semester* function becomes $O(V + E)$. $\square$

Space Complexity. Space complexity is determined by size of *adjList* adjacency list, *indegree* array, *sem* array and q queue.In worst case maximum size of queue is of $O(V)$, overall space complexity $= O(V + E) + O(V) + O(V) + O(V) = O(V + E)$ $\square$

2.3 2.3

2.3

Question. *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 $P \subseteq C \times C$ of all those pairs (c, c') for which the intersection $L(c) \cap L(c')$ is empty.*

Solution. We propose the following algorithm to compute the set P:

Algorithm 6 Computing set P

procedure COMPUTEP(G)
  $G' \leftarrow reverse(G)$ ▷ reverse all edges in G
  $L \leftarrow$ empty map
  for all $c \in C$ **do**
    $L(c) \leftarrow DFS(G, c)$ ▷ Utility DFS function which returns list of nodes that can be visited by c
    $L(c) \leftarrow sort(L(c))$ ▷ Sort the elements in $L(c)$ using counting sort
  end for
  $P \leftarrow \{\}$
  for all $(c_1, c_2) \in C \times C$ **do**
    if $L(c_1) \cap L(c_2) = \phi$ **then**
      $P \leftarrow add(P, (c_1, c_2))$
    end if
  end for
  return P
end procedure

We assume that the given graph G is a DAG.
After we reverse the edges in G, we have edges from a course c to its pre-requisites and the new graph G' is also a DAG. Therefore, running DFS on this graph, we will reach all courses that need to be completed before we can credit course c. This can be shown recursively to be true since to credit c, we need to credit all its children first and then we can credit c (structural induction). Therefore we can perform a DFS from each node c in G' to obtain $L(c)$.
Now, since there are only n nodes in G', we can map each node to a number from $\{1, 2, \ldots, n\}$ and thus on performing counting sort on $L(c)$, we can efficiently compute the sorted version of $L(c)$ (complete analysis will be done along with computing time complexity). We then iterate over all pairs (c_1, c_2) in the graph G' and compute the intersection of $L(c_1)$ and $L(c_2)$. If this intersection is ϕ, we add (c_1, c_2) to the set P.

Space Complexity: $L(c)$ will have $O(n)$ elements and since we compute $L(c)$ for all nodes, L will take a space of $O(n^2)$. The set P will have atmost $O(n^2)$ elements. The space used everywhere else is $O(1)$. Therefore the total space complexity of Algorithm 6 is $O(n^2)$.
Time Complexity:

1. Reversing the edges in G can be done in $O(n + m) = O(n + n^2) = O(n^2)$ time since we

are using adjacency list format to represent the graph G.

2. DFS for each node takes $O(n+m) = O(n^2)$ time. Since there are n nodes, running DFS for all nodes takes $O(n^3)$ time.

3. Sorting each $L(c)$ can be done in $O(size(L(c)) + \max(L(c))) = O(n+n) = O(n)$ time. Therefore sorting $L(c)$ for all $c \in C$ will take $O(n^2)$ time.

4. Iterating over all pairs of (c_1, c_2) will involve $O(n^2)$ iterations. Computing intersection of $L(c_1)$ and $L(c_2)$ can be done in $O(size(L(c_1)) + size(L(c_2))) = O(n+n) = O(n)$ time since all $L(c)$ are sorted (using a strategy similar to that of merging two arrays). Therefore computing P will take $O(n^3)$ steps.

Therefore the total time complexity of Algorithm 6 is $O(n^3)$.

Thus, we have proposed an $O(n^3)$ solution to compute set P and have also argued its correctness. $\square$

3 Question 3

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 — $c_1, \ldots, c_n$, 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.

3.1 3.1

Question 3.1

Question. *Design an algorithm to verify whether or not there exists a cycle* $(c_{i_1}, \ldots, c_{i_k}, c_{i_{k+1}} = c_{i_1})$ *such that exchanging money over this cycle results in positive gain, or equivalently, the product* $R(i_1, i_2) \cdot R(i_2, i_3) \cdots R(i_{k-1}, i_k) \cdot R(i_k, i_1)$ *is larger than* 1.

Solution. Consider the transformation function $t : \mathbb{R}^+ \to \mathbb{R}$,

$$t(x) = -\log(x) \tag{6}$$

We observe that t is a strictly decreasing function. Applying this tranformation function on the edges in the given graph G (the function can be applied since all edges $R(i, j)$ are positive), we get the condition for *positive gain* as:

$$\begin{aligned} & R(i_1, i_2) \cdot R(i_2, i_3) \cdots R(i_{k-1}, i_k) \cdot R(i_k, i_1) > 1 \\ \iff & t(R(i_1, i_2) \cdot R(i_2, i_3) \cdots R(i_{k-1}, i_k) \cdot R(i_k, i_1)) < t(1) \\ \iff & \sum_{j=1}^{k} t(R(i_j, i_{j+1})) < 0 \end{aligned} \tag{7}$$

Therefore, after applying the transformation on the edges, our problem reduces to finding a cycle of negative weight in the graph G.
We assume that the given graph is strongly connected, since it makes sense to be able to perform currency exchange between any two currencies i, j. This ensures that all cycles present in G are accessibe from any vertex i. To find a cycle of negative weight, we will run Bellman Ford algorithm for $|V| = n$ iterations and if there is a change in the distances between any pair of the vertices in the last iteration, we will report detection of cycle with negative weight.

Algorithm 7 Detect cycle with negative weight

```
1: procedure DetectCycle(G)
2:     u ← any vertex of G
3:     n ← |V|
4:     d ← array of size n initialised with ∞
5:     d[u] ← 0
```

```
6:      for i in [1, n − 1] do
7:          for all (x, y) ∈ E do
8:              if d[y] > d[x] + t(R(x, y)) then
9:                  d[y] ← d[x] + t(R(x, y))
10:             end if
11:         end for
12:     end for
13:     for all (x, y) ∈ E do
14:         if d[y] > d[x] + t(R(x, y)) then
15:             return True
16:         end if
17:     end for
18:     return False
19: end procedure
```

To prove the correctness of the above algorithm, we will prove the following claim:

Claim 3.1. *There exists a walk of shortest length from a source vertex u to a vertex v using atmost $n-1$ edges if there is no cycle of negative weight.*

Proof. Let G have no cycle of negative weight. Now, consider a walk W of shortest length using more than $n-1$ edges. If no such walk exists then we have trivially proved the claim. Since W uses more than $n-1$ edges, there exists a vertex w which appears atleast twice in W. Let the walk and its length be given as:

$$\begin{aligned} W &= \{u, \ldots, w, x_1, \ldots, x_k, w, \ldots, v\} \\ \implies d(W) &= \mathit{length}(\{u, \ldots, w\}) + \mathit{length}(\{w, x_1, \ldots, x_k, w\}) + \mathit{length}(\{w, \ldots, v\}) \end{aligned} \tag{8}$$

Now consider the length of walk W' given by $\{u, \ldots, w, \ldots, v\}$ (after removing the cycle at w):

$$\begin{aligned} & d(W') = \mathit{length}(\{u, \ldots, w\}) + \mathit{length}(\{w, \ldots, v\}) \\ \implies & d(W') = d(W) - \mathit{length}(\{w, x_1, \ldots, x_k, w\}) \\ \implies & d(W') \leq d(W), \text{ no negative cycle exists in } G \\ \implies & d(W') = d(W), \; d(W) \text{ is the shortest length of any walk} \end{aligned} \tag{9}$$

We know that $|E(W')| < |E(W)|$ and thus we have found another walk of shortest length using lesser number of edges than W. Note that W' might still use more than $n-1$ edges. Now, we repeat the above procedure to obtain a walk W_0 such that no vertex in W_0 appears twice. Therefore, $|E(W_0)| \leq n-1$. Therefore, we have found the walk of shortest length using atmost $n-1$ edges. This completes the proof of the claim. □

Now, if G has no cycle of negative weight, then Algorithm 7 will have no improvement in the last `for` loop, using Claim 3.1. Additionally, by the contrapositive of Claim 3.1, there will be an update in the n^{th} iteration given by lines $13-18$ of Algorithm 7.

Time and Space Complexities: The time and space complexities of the algorithm are identical to those of the Bellman-Ford algorithm. The time complexity is $O(n \times m)$ and the space complexity is $O(n)$. □

3.2 3.2

Question 3.2

Question. *Present a cubic time algorithm to print out such a cyclic sequence if it exists.*

Solution. We will modify the solution proposed in Question 3.1 to also return the cycle of negative weight if one exists.

Algorithm 8 Detect and return cycle with negative weight

```
procedure FindCycle(G)
    u ← any vertex of G
    n ← |V|
    d ← array of size n initialised with ∞                ▷ distance array
    p ← array of size n initialised with null              ▷ parent array
    d[u] ← 0
    for i in [1, n − 1] do
        for all (x, y) ∈ E do
            if d[y] > d[x] + t(R(x, y)) then
                d[y] ← d[x] + t(R(x, y))
                p[y] ← x
            end if
        end for
    end for
    v ← null
    for all (x, y) ∈ E do
        if d[y] > d[x] + t(R(x, y)) then
            d[y] ← d[x] + t(R(x, y))
            p[y] ← x
            v ← y
        end if
    end for
    if v = null then
        return null
    end if
    for i in [1, n] do
        v ← p[v]
    end for
    C ← {v}
    w ← p[v]
```

```
31:     while w ≠ v do
32:         C ← add(C, w)
33:         w ← p[w]
34:     end while
35:     C ← reverse(C)
36:     return C
37: end procedure
```

To prove the correctness of Algorithm 8, we notice that the algorithm is the same until line 22 except for the addition of array p and maintaining an additional variable v. The array p generates the *parent* array which denotes that the shortest path from u to any vertex v is exactly equal to the shortest path from u to $p[v]$ and the edge $(p[v], v)$. This is trivially true using the construction of the Bellman-Ford algorithm.
Now, we find any vertex v which can be reached from u after passing through a cycle of negative weight. If no such vertex is found then we report the same. Else, a negative cycle exists and we proceed to find it using the algorithm from lines $26 - 35$. We will now prove the following claim which will complete the correctness proof of Algorithm 8:

Claim 3.2. *The lines* $26 - 35$ *finds a valid cycle in* G *if one exists.*

Proof. We know that the path from u to v passes through a cycle of negative weight since the distance between u and v decreased in the n^{th} iteration. Now consider the vertex set C of this cycle. The parent of each of vertex $c \in C$ also lies in C. Assume by contradiction that this is not the case. Then there exists a vertex $x \in C$ such that $p[x] \notin C$. Now, consider the path from u to v. It is given by:

$$path(u, v) = path(u, x) \cup path(x, y) \cup path(y, v) \tag{10}$$

Here, y is the first vertex which is a part of cycle C and is obtained by traversing on the ancestors of v $(\{v, p[v], p[p[v]], \ldots\})$. Now, this path doesn't pass through any cycle of negative weight since C isn't a part of this path. This is a contradiction to our assumption that exists a vertex $x \in C$ such that $p[x] \notin C$. Therefore $\forall c \in C, p[c] \in C$.
Now, consider the `for` loop on lines $26-28$, we iterate n times on the descendents of v. This ancestor v is now a part of the cycle since G has n vertices and after we reach y, all ancestors lie in C. Since y is reached in $\leq n$ steps, the final vertex $v \in C$.
After we have arrived on the cycle C, we proceed to retrieve it. For this, we iterate backwards on the parent of the vertices in C until we reach the starting vertex v (lines $31 - 34$). The vertices added into C are in the reverse order since we are iterating from child to parent and thus we reverse the set C and return it.
Therefore we have shown that Algorithm 8 finds a valid cycle in G if one exists. This completes the proof of the claim. □

Thus, using Claim 3.2 we have shown the correctness of Algorithm 8. We will now compute the time and space complexities.

Time and Space Complexities: We use an additional space of $O(n)$ to store p and the cycle C and therefore the total space complexity is $O(n)$. To analyse the time complexity, we observe that the additional computation that we perform in lines $26 - 36$ are all bounded by $O(n)$ since we cannot have a cycle of length larger than n. Therefore the total time complexity is given by $O(n \times m) = O(n \times n^2) = O(n^3)$.

Therefore we have proposed a cubic time algorithm to retrive a cycle with negative weight in Algorithm 8 and have also argued its correctness. $\square$

4 Question 4

4.1 4.1

4.1

Question. *You are given a set of k denominations. Devise 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, $d[1], \ldots, d[k]$.*

Solution. The assumptions made are that the number of coins of every denomination are infinite and they are integral values.
We solved this problem using dynamic programming. Given the cost n and array of possible denominations $denom$ with size k, we create $dpTable$ which is an $(n+1)$ array. $dpTable[i]$ counts the number of ways to generate value i using the given denominations. The answer is obtained by observing value of the last element $dpTable[n]$.

Algorithm 9 Find total possible combinations of denominations to achieve value of n

```
procedure COMBINATIONS(denom, n)
    k ← size(denom)                                  ▷ number of types of denominations
    dpTable ←1D-zero array of size (n + 1)
    dpTable[0] ← 1                      ▷ there is trivially one way to generate sum 0
    for j in [1, k] do
        for i in [denom[j], n] do
            dpTable[i] ← dpTable[i] + dpTable[i − denom[j]]
        end for
    end for
    return dpTable[n][k]
end procedure
```

Proof of correctness. We will prove the correctness of the transitions in $dpTable$ using induction on the denominations used.
Base case: $j = 0$ is true since there is trivially one way to generate sum 0 using no denominations and no other value can be generated.
Inductive step: Assume the claim is true for $j - 1$, consider denomination j: We have generated the number of ways we can generate all values using the first $j - 1$ denominations. Now, consider the j^{th} denomination.
We perform another induction on the values i. The base case for $i = denom[j] - 1$ is trivially true (there is no way to generate the sum using the j^{th} denomination). Assume that we have generated all valid combinations for the first $i - 1$ values using the first j denominations. Now, we can generate value i using the first $j - 1$ denominations (the value currently in $dpTable[i]$) or we can use the first j denominations. To use the j^{th} denomination, we need to use $denom[j]$ atleast once. Therefore, we need to find the number of ways we can generate $i - denom[j]$ using the first j denominations. This is already computed by our inductive assumption. Therefore, the number of ways to generate i using the first j denominations is

given by:

$$dp(i) = dp(i) + dp(i - denom[j]) \tag{11}$$

This completes the induction on the value i.
We have shown that number of ways to generate all values is done correctly in the j^{th} step. Therefore, we have proved the correctness of the inductive claim for j. This completes the induction and proof of correctness for Algorithm 9. □

Proof of termination. Here, we have a finite table of size $(n + 1)$. We iterate through the entire table and exit successfully in any case. Hence the algorithm terminates. □

Time Complexity. Deciding factors for time-complexity in big-Oh notation are going through the entire *dpTable* and running a for loop with k iterations at each index of the table. Time complexity $= O(n \times k)$
This is a polynomial time solution. □

Space Complexity. We create a *dpTable* of size $n+1$ and use constant space everywhere else. Space complexity $= O(n)$ □

□

4.2 4.2

4.2

Question. *You are given a set of k denominations. Device a polynomial time algorithm to find a change of Rs.n using the minimum number of coins.*

Solution. The assumptions made are that the number of coins of every denomination are infinite and they are integral values.

Algorithm 10 Find minimum number of denominations to achieve value of n

```
procedure LeastCurr(denom, n)
    k ← size(denom)                                    ▷ number of types of denominations
    dpArr ← array of size n + 1 initialised with ∞
    dpArr[0] ← 0                                       ▷ Base case: no coin needed n = 0
    for index in [1, n] do
        for i in [1, k] do
            if index − denom[i] ≥ 0 then
                dpArr[index] ← min(dpArr[index], dpArr[index − denom[i]] + 1)
            end if
        end for
    end for
    return dpArr[n]
end procedure
```

□

Proof of correctness. We will prove the correctness of Algorithm 10 using induction on the value i.
Base case: $i = 0$ is true since we can generate a sum 0 using 0 coins.
Inductive step: Assume that the claim is true for $i - 1$. Consider the sum i
The last denomination used to generate the sum i has to be one of the denominations $denom[j]$ such that $i \geq denom[j]$. Therefore, the minimum number of coins needed to generate i will be one more than the least number of coins needed to generate the sum $i - denom[j]$ for all valid j. Therefore the transition equation is given by:

$$dp(i) = 1 + \min_{j | i \geq denom[j]} dp(i - denom[j]) \tag{12}$$

Therefore we have shown the claim to be true for i and this completes the proof of induction and hence the correctness of Algorithm 10

Proof of termination. We iterate through the entire array of size n and exit successfully in any case. Hence the algorithm terminates. □

Time Complexity. Deciding factors for time-complexity in big-Oh notation are going through the entire *dpArray* and running a for loop of k iterations for each index.
Time complexity $= O(n \times k)$
This is a polynomial time solution. □

Space Complexity. We create a *dpArray* of size $n+1$ and use constant space everywhere else.
Space complexity $= O(n)$ □

□