COL351 Assignment 2

Rahul Chhabra
2019CS11016

Shrey J. Patel
2019CS10400

September 27, 2021

1. Algorithms Design book

Alice, Bob, and Charlie have decided to solve all exercises of the Algorithms Design book by Jon Kleinberg, Eva Tardos. There are a total of n chapters, [1,..., n], and for i $\in$ [1, n], x_i denotes the number of exercises in chapter i. It is given that the maximum number of questions in each chapter is bounded by the number of chapters in the book i.e. n. Your task is to distribute the chapters among Alice, Bob and Charlie so that each of them gets to solve nearly an equal number of questions. Device a polynomial time algorithm to partition [1,..., n] into three sets S1, S2, S3 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:
It is given that there are n books numbered as [1,..., n] and arr_i represents the number of chapters in the i^{th} book. We have to divide the book among three sets S_1, S_2 and S_3 to minimize max($\sum_{i\in S_1} x_i$,$\sum_{i\in S_2} x_i$,$\sum_{i\in S_3} x_i$). Since this is an optimisation problem having an optimal substructure discussed below, we can use dynamic programming to solve this problem.

DP Table: Let **dp[n + 1][sum + 1][sum + 1]** be a three dimensional boolean array where sum represents the sum of all elements in the given array and n represents the number of elements in the array.

DP State: dp[i][j][k] represents whether we can partition first i books into three sets such that sum of all elements in one set is j and sum all elements in any of the remaining set is k. Without loss of generality, let the first set be S_1 and second set be S_2. So $\sum_{i\in S_1} arr_i$ = j and $\sum_{i\in S_2} arr_i$ = k.

Recurrence Relation:

$$OPT(i,j,k) = \begin{cases} True, & \text{if } j == 0 \text{ and } k == 0 \\ False & \text{if } i < 0 \text{ or } j < 0 \text{ or } k < 0 \\ OPT(i-1,j,k) \vee OPT(i-1,j-arr[i],k) \vee OPT(i-1,j,k-arr[i]), & \text{otherwise} \end{cases}$$

Working Python Code for reference(Click on the icon below):

Justification for correctness:

Base Case: j==0 and k==0
We can show that this case is always true since we can assign all the elements to set S_3 such that S_1 and S_2 will be empty. Also even if there are no books to assign (i<0) we can have a solution with all sets empty. Therefore base case holds.

Justification for recurrence: Suppose we have answer for dp[i-1][_][_], then to calculate any dp[i][j][k], we can add i^{th} book into any of the three sets, and check if we can get S_1 and S_2 such that $\sum_{i \in S_1} arr_i$ = j and $\sum_{i \in S_2} arr_i$ = k

- **Case 1:** i^{th} book is added to set A, our goal then is to check whether there is a partition that follows the above condition and i^{th} book is inserted in S_1. If such a partition exists, we should be able to remove the i^{th} book from S_1 and the corresponding partition should also exist i.e. we should check whether dp[i-1][j - arr[i]][k] is true or not.
- **Case 2:** i^{th} book is added to set B, our goal then is to check whether there is a partition that follows the above condition and i^{th} book is inserted in S_2. If such a partition exists, we should be able to remove the i^{th} book from S_2 and the corresponding partition should also exist i.e. we should check whether dp[i-1][j][k - arr[i]] is true or not.
- **Case 3:** i^{th} book is added to set C, then we need to check whether there is a partition that follows the above condition and i^{th} book is inserted in S_3. If such a partition exists, we should be able to remove the i^{th} book from S_3 and the corresponding partition should also exist i.e. but since our DP state only mentions about the sum of elements in S_1 and S_2, we should check whether dp[i-1][j][k] is true or not.

If any of the above case is true, we can say that it is possible to get partition such that two subsets S_1 and S_2 have sum j and k respectively. So, dp[i][j][k] is also true.

Now we need to find a partition 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. To calculate this, we can calculate all the possible set sums with n books by traversing the dp array for i = n and store the minimum value of max($\sum_{i \in S_1} x_i$,$\sum_{i \in S_2} x_i$,$\sum_{i \in S_3} x_i$) and the corresponding set sums. This step returns the optimal value due to correctness of DP Table values as argued above.

For returning the books in each set, we need to traverse the DP table starting with i = n from the cell containing optimal values of j and k and i $\geq$ 0. At each step we can have three possibilities:

- Case 1 : (j-arr[i]>=0) and dp[i-1][j-arr[i]][k] is True
  In this case, we can conclude that the i^{th} book was inserted in set A, so we can put this book in set A and set i $\leftarrow$ i-1 and j $\leftarrow$ j-arr[i].
- Case 2 : (k-arr[i]>=0) and dp[i-1][j[k-arr[i]] is True
  In this case, we can conclude that the i^{th} book was inserted in set B, so we can put this book in set B and set i $\leftarrow$ i-1 and k $\leftarrow$ k-arr[i].
- Case 3 : Since the i^{th} book was not inserted in set A and B, it must be inserted in set C. So, we can put this book in set C and set i $\leftarrow$ i-1.

Time Complexity Analysis:
To get the optimal partition we need to fill the DP Table completely. The total number of values to be calculated is n * total_sum * total_sum and to calculate each value, we require O(1) computations since we already have already stored the solved sub-problems.
Also it is given that the number of chapters in any book is less than or equal to the total number of available books i.e. n
$\implies arr_i \leq n \ \forall \ i \in [1, n]$
$\implies \sum_i arr_i \leq n^2 \ \forall \ i \in [1, n]$
$\implies$ total_sum $\leq n^2$
So the time complexity of above algorithm will be O(n*total_sum*total_sum) = O(n * n^2 * n^2) = O(n^5)

which is polynomial time in n.

Space Complexity Analysis :
We would need to store the entire DP Table since we want to return contents for each set.So, the space complexity will be proportional to the number of elements in the DP Table i.e O(n*total_sum*total_sum) = O(n^5)

2. Course Planner

You are given a set C of courses that needs to be credited to complete graduation in CSE from IITD. Further, for each c ∈ C, you are given a set P(c) of prerequisite courses that must be completed before taking the course c.

1. Device the most efficient algorithm to find out an order for taking the courses so that a student is able to take all the n courses with the prerequisite criteria being satisfied, if such an order exists. What is the time complexity of your algorithm?
2. Device the most efficient algorithm to find minimum number of semesters needed to complete all n courses. What is the time complexity of your algorithm?
3. Suppose for a course c ∈ 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 a pair set P ⊆ C × C such that for any (c, c') ∈ P, the intersection L(c) ∩ L(c') is empty.

Graph Formulation:
Before solving any of the parts, we will formulate the given problem as a graph. Consider the graph G = (V,E), where V = set of all the courses required for graduation and E = $\{(c', c) \mid c' \in P(c)\ \forall\ c \in V\}$, i.e. every directed edge (x,y) is such that x is a pre-requisite course of y. Also, earlier, *P(c)* = set of all pre-requisites of course *c*.

Observation 1: If some course c' is an indirect pre-requisite i.e. ancestor of c $\iff$ c is reachable from c' in the graph formulated
Proof of observation 1:
If c' is an ancestor of c then ∃ a sequence of courses $c_0 = c', c_1, c_2, ..., c_n = c$ such that $c_i \in P(c_{i+1})\ \forall\ 0 \leq i < n$. And each pair (P(c),c) is an edge in E. So, $c_i \to c_{i+1} \in E\ \forall\ 0 \leq i < n$. So, there is a path $c_0 = c', c_1, c_2, ..., c_n = c$ in G. So, c is reachable from c'.

Observation 2: G is acyclic
Proof of observation 2:
Proof by contradiction. Say, there exists a cycle $c_0, c_1, ...c_n, c_0$. So, ∃ a path P from c_0 to c_n. So, from observation 1, c_0 is a pre-requisite of c_n. But there is also a path P' from c_n to c_0, which means c_n is a pre-requisite of c_0. But pre-requisiteness is an anti-symmetric relation. So, we have a contadiction. So, G is acyclic.

Solution for (a):

Algorithm:

1. We will use the following variables:
   (a) The required sequence as A. Initially, A = ϕ.
   (b) A global ***timer*** variable to be used in DFS, which is zero initially.
   (c) We will assume that set of vertices V is arbitrarily ordered and indexed.
   (d) A boolean array, with the same indexing on vertices as V, named ***visited***, which checks if a vertex has been visited during DFS or not.
   (e) Two integer arrays ***start*** and ***finish***, with the same indexing as V, to store the start and finish times of DFS of a particular vertex.

2. First, choose the first unvisited vertex, v from the set V.
3. Invoke *DFS(v)* to traverse all the vertices reachable from v in the graph.
4. In DFS(v), we mark the vertex v as visited, assign the current value of *timer* to *start[v]* and increment the timer. Now, we recursively invoke DFS on the unvisited out-neighbours of v.
5. At the end of the DFS call, we assign the current value of *timer* to *finish[v]*, increment the timer and add vertex v to the end of A before returning.
6. Repeat steps 2 to 5 until there is no unvisited vertex in V, i.e. |A| = |V|.
7. Reverse the sequence A. The resulting sequence gives an ordering of vertices which the student can follow under the given constraints.

Algorithm 1 Pseudocode for finding a feasible ordering of vertices of G

```
timer ← 0
visited ← boolean array of size |V| with all entries false
start ← integer array of size |V|
finish ← integer array of size |V|
A ← φ

procedure findOrdering(V,E):
    i ← 0
    while i < |V| do
        if visited[i] then
            i ← i+1
        else
            DFS(v)
            i ← i+1
        end if
    end while
    A ← reverse(A)
    return A
procedure DFS(v)
    visited[v] ← true
    start[v] ← timer
    timer ← timer + 1
    for all out-neighbours u of v do
        if visited[v] then
            continue
        else
            DFS(u)
        end if
    end for
    finish[v] ← timer
    timer ← timer + 1
    A ← A.insert(v)
```

Time Complexity Analysis:
In the above algorithm, each vertex is visited once and also, each edge is explored at most one, and so the total complexity of exploring the graph and assigning start and finish times is **O(m+n)**. Further, time complexity of reversing the final sequence is **O(n)**. So, total time complexity = **O(m+n)**.

Space Complexity Analysis:
The only extra space used is A itself, which is of size n, so total space complexity is O(n).

Proof of correctness:

Claim 2.1.1: The final sequence contains vertices in descending order of their finish times.
Proof of Claim 2.1.1: Consider this sequence, A before reversal. Then it is sufficient to prove that A has all the vertices in ascending order of their finish times. Note that, according to the algorithm, a vertex is added only after assigning the current value of timer to the finish value of that vertex and the timer value is incremented after that. So, if some vertex u is added before v to A, *finish[u] < finish[v]*. And any new vertex is always appended to the end of A. So, A is sorted in increasing order, and consequently its reverse is sorted in decreasing order of finish times.

Now we need to prove that the sequence A is feasible, i.e. for any course, its pre-requisite appears before itself in the ordering denoted by A, which is the decreasing order of finish times.

Claim 2.1.2: If some course c' is a pre-requisite of c, then *finish(c') > finish(c)*.
Proof of claim 2.1.2: The proof is trivial for a vertex which has no pre-requisites i.e. no in-neighbours. Otherwise, from observation 1, there is a path P in G from c' to c. We will prove this by considering two cases:

1. **Case 1:** c' is visited before c. Since there is a path from c' to c, c is reachable from c'. From the property of DFS traversal, all descendants of c' should be explored before DFS(c') returns. So, exploration of c finishes before that of c'. So, *finish(c) < finish(c')*.
2. **Case 2:** c is visited before c'. We know from observation 2 that G is acyclic. So, there is no path from c to c', or in other words, c' is not reachable from c. So, DFS(c) doesn't explore c'. Thus, DFS(c) returns before c' is visited. So, exploration of c ends even before c' is visited. So, *finish(c) < finish(c')*.

The claim holds from the above two cases. From claim 2.1.1 and 2.1.2, we have that, if we explore the courses in the sequence A, then no course will be taken before its pre-requisite. Hence Proved.

Solution for (b):

We will re-use the ordering A obtained in part(a) for this problem. We will start by using some properties before constructing the algorithm.

Claim 2.2.1: The number of semesters required should be at least equal to the length of the longest path in the graph.

Proof of claim 2.2.1:
Let *sem[c]* denote the semester in which course c is to be taken. We know from the graph formulation, that all the vertices on any path form a strict partial order(since we assume that a course and its pre-requisite cannot be done in the same semester), so if c' is a pre-requisite of c, then *sem[c']* < *sem[c]*.

We will prove this claim by contradiction. Suppose that the number of semesters required are less than the longest path of the graph, P = $c_1, c_2, ...c_n$. So, *sem[c]* ∈ [1,2,...m] where m<n. From above observation, all the vertices on path P should have distinct semester values. In other words, we need to assign n objects(courses) in m buckets(semesters) where n>m. Then, from pigeonhole principle, one of the buckets will have at least two objects. So, there is a pair (c,c') such that $c, c' \in P$ and *sem[c]* = *sem[c']*.

This clearly violates the above observation. So, by contradiction, we have that $m \geq n$, which is to say that the number of semesters should be at least equal to the length of the longest path in the graph.

Now, from claim 2.2.1, finding the minimum number of semesters required is same as finding the length of the longest path in the graph. We will use dynamic programming to solve this problem.

DP formulation:
Let len[v] denote the length of the longest path in G starting from v. Now, these length values are related to each other by the following recurrence:

Recurrence Relation:

$$len[v] = \begin{cases} 0, & S_v = \phi \\ 1+\max_{s \in S_v} len[s], & \text{otherwise} \end{cases}$$

where S_v is defined as $\{s \in V \mid (v, s) \in E\}$

Algorithm:

1. Obtain the ordering A from the algorithm 1 and reverse it.
2. Now, visit the next vertex v of the sequence A, and calculate len[v] using the above recurrence.
3. Maintain a variable *maxLength* which stores the length of the longest path found in the graph explored so far.
4. Repeat steps 2 and 3 until all the vertices of A have been processed.

Algorithm 2 Pseudocode for finding the length of the longest path in a graph

```
procedure longestPath(V,E)
  A ← findOrdering(V,E)
  A ← reverse(A)
  len ← an array of size |V|
  maxLength ← 0
  i ← 0
  for all vertices v in A do
      if outdegree(v) = 0 then
          len[v] ← 0
      else
          len[v] ← 0
          for all out-neighbours u of v do
              len[v] ← max(len[v],len[u])
          end for
          len[v] ← len[v] + 1
      end if
      maxLength ← max(maxLength,len[v])
  end for
  return maxLength
```

Proof of correctness:
Before moving to the main proof statement, we prove a small claim.

Claim 2.2.2:
In the ordering A, all the descendants of a node will always be visited before itself.

Proof of claim 2.2.2:
Here, note that A is the reverse of the ordering obtained from part (a), i.e. increasing order of finish times. We proved for that the descending order of finish times is such that the ancestor of any vertex will always be visited before the vertex itself. So, in reverse order, any pre-requisite(ancestor) nodes of a given node will always be visited after itself. (Reversal of strict ordering. In other words, for any node, all its descendants will always be visited before itself.

We will prove this by induction on the length of A. Let A = $\{v_1, v_2, ...v_n\}$. We will use the following hypothesis:
H(i): len[v] = length of the longest path starting from v in the graph formed by A_i, $\forall v \in A_i$, where $A_i = \{v_1, v_2, ...v_i\}$.

- **Base Case:** i=1. In this case, only len[v_1] will be calculated. Since v_1 is the first vertex to be visited, it has no descendants from claim 2.2.2. So, $S_{v_1} = \phi$ and thus from recurrence relation, len[v_1] = 0. This is consistent with the observation since the graph formed by $A_1 = \{v_1\}$ has only one path v_1 of length 0. So, the base case holds.
- **Induction Step:** Assume that the hypothesis holds for A_{i-1}. Now, we need to prove that it also holds for $A_i = A_{i-1} \cup v_i$. From hypothesis, all the len[v] values are correct $\forall\ v \in A_{i-1}$. Now, for v_i, we have two cases:

  1. v_i has no descendant in G. From recurrence relation, len[v_i] = 0. In this case, no vertex in A_{i-1} is reachable from v_i and so the longest path in A_i starting from v_i is v_i itself, whose length is 0. So, this is consistent with the recurrence relation.

2. v_i has some descendants in G. Then, from claim 2.2.2, all such descendants have already been explored i.e. $S_{v_i} \subseteq A_{i-1}$. From induction hypothesis, len[v] is correct $\forall v \in S_{v_i}$. And from recurrence relation, len[v_i] = 1 + $\max_{v \in S_{v_i}} len[v]$. This is consistent with the recursive definition of the longest path, that the length of the longest path from v_i should be one more than the length of the longest path of all of its children.

   So, the hypothesis holds for both the cases. Hence, the claim holds.

And, according to the algorithm, maxLength contains the length of the longest path in the entire graph i.e. $A_n = A$, where:

$$maxLength = \max_{v \in V} \text{len[v]}$$

Analysis:

- **Time Complexity:** First of all we need to find the ordering from part(a) which requires O(m+n) time.Now, for each vertex, the len value is calculated at most once, and every calculation requires O(1) computations. So, total time complexity = **O(m+n)**.
- **Space Complexity:** The only extra space utilized in the algorithm is for storing the len values for each of the vertex, and so auxiliary space is proportional to the number of vertices. So, total space complexity = **O(n)**.

Solution for (c):

Algorithm:

- For each node in graph, do a DFS and store all the nodes reachable from it in a Hash Table (Set).
- Store all possible pair of nodes in a set.
- For each pair of nodes (u,v), traverse all the nodes and check whether the current node is a common ancestor of both u and v. Suppose the current node is c, if c is an ancestor of u as well as v, then remove (u,v) and (v,u) from the set of all possible pairs.
- The set remaining after the above operation for each pair of node will be the answer.

We will use the following lemma proved in Lecture 8

> Lemma 1: The vertices visited during recursive call DFS(x) are descendants of x in DFS tree.

Claim 2.3.1: The set returned by DFS(u) is the set of all descendants of u.
Proof: By Lemma 1

Claim 2.3.2: isAncestor(p,c) return true if and only if p is an ancestor of c in the original graph.

Proof of Claim 2.3.2: Let's first assume that for some p,c ∈ V, isAncestor(p,c) returns true but p is not an ancestor of c in the original graph. Since DFS(p) correctly returns the descendants of p and isAncestor(p,c) returns true $\implies$ c ∈ Descendants[p]. This means that there is a path from p to c in the original graph which contradicts the fact that p is not an ancestor of c.

Algorithm 3 Pseudocode for generating set S

```
descendant_set_array ← an array storing set of descendants for each node
 procedure generateSet(V,E):
   result ← φ
   for vertex u in V do
       for vertex v in V do
           if u != v then
               result.insert((u,v))
           end if
       end for
   end for
   for vertex a in V do
       descendant_set_array[a] = DFS(a)
   end for
   for vertex u in V do
       for vertex v in V do
           for vertex p in V do
               if isAncestor(p,u)  isAncestor(p,v) then
                   result.remove((u,v))
                   result.remove((v,u))
               end if
           end for
       end for
   end for
   return result
procedure isAncestor(p,c):
   if descendant_set_array[p].contains(c) then
       return true
   else
       return false
   end if
procedure DFS(u):
   visited ← an array of size |V|, all initialized to false
   descendants_set ← φ
   DFSUtil(u, *visited, *descendants_set)
   return descendants_set
procedure DFSUtil(u, visited, descendants_set):
   visited[u] = true
   descendants_set.insert(u)
   for all out-neighbours v of u do
       if !visited[v] then
           DFSUtil(v, visited, descendants_set)
       end if
   end for
```

Now let's assume that for some p,c $\in$ V, p is an ancestor of c but isAncestor(p,c) returns false. Since isAncestor(p,c) returns false $\implies$ c $\notin$ Descendants(p). This means that there is no path from p to c in the original graph, which contradicts the fact that p is an ancestor of c.

Claim 2.3.3 : Let S be the set returned by above algorithm,then any pair of nodes (a,b) $\in$ S if and only if L(a) $\cap$ L(b) = ϕ

.

Proof of Claim 2.3.3:
We will first show that (a,b) $\in$ S $\implies$ L(a) $\cap$ L(b) = ϕ
Let us assume that there exists a pair (a,b) $\in$ S such that L(a) $\cap$ L(b) is non-empty. Let's take any c such that c $\in$ L(a) $\cap$ L(b). So by definition of list L, $\exists$ a path from c to a as well as from c to b. This also means that c is an ancestor of both a and b i.e. both isAncestor(c,a) and isAncestor(c,b) are true, so our algorithm such pair from S. This contradicts the fact that (a,b) $\in$ S. So we can conclude that (a,b) $\in$ S $\implies$ L(a) $\cap$ L(b) = ϕ.

Now we will show that L(a) $\cap$ L(b) = ϕ $\implies$ (a,b) $\in$ S
Let us assume to the contrary that there exists a pair (a,b) such that L(a) $\cap$ L(b) = ϕ but (a,b) $\notin$ S. Since L(a) and L(b) denotes the sets of ancestors of a and b respectively, L(a) $\cap$ L(b) = ϕ $\implies$ there is no common ancestor between a and b. So the proposition (isAncestor(c,a) $\wedge$ isParent(c,b)) is false. So our algorithm cannot remove (a,b) and (b,a) from the resulting set. But this contradicts the fact that (a,b) $\notin$ S. So, (a,b) $\in$ S.

So our algorithm returns the required set S.

Time Complexity Analysis :

1. Time Complexity of DFS function : O(m + n). Since we are doing DFS for all the n nodes, the contribution of DFS to time complexity will be O(n*(m+n)) and assuming that m = O(n^2), the total time complexity of DFS from all nodes is O(n^3).
2. Time complexity of isAncestor function : O(1) since we use hash table for checking whether a node is an ancestor of any other node.
3. In the generateSet function, we have three for loops each of which runs over all the n vertices, so there will be a total of n^3 iterations, each of which takes O(1) time.

Overall time complexity of algorithm : O(n^3) + O(1) + O(n^3) = O(n^3)

Space Complexity Analysis :

For each node we are storing all its descendants, and since each node can have O(n) number of descendants, overall space complexity of above algorithm = O(n^2)

3. Forex Trading

Suppose you are a trader aiming to make money by taking advantage of price differences between different currencies. You model the currency exchange rates as a weighted network, wherein, the nodes correspond to n currencies - $c_1, c_2, ..., c_n$, 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.

1. Design an algorithm to verify whether or not there exists a cycle $(c_{i_1}, ..., c_{i_k}, c_{i_k+1} = c_{i_1})$such that exchanging money over this cycle results in positive gain, or equivalently, the product $\mathrm{R}[i_1, i_2] \cdot \mathrm{R}[i_2, i_3] \cdot \cdots \mathrm{R}[i_{k1}, i_k] \cdot \mathrm{R}[i_k, i_1]$ is larger than 1. (Hint: Use the fact that a number x is strictly larger than 1 if and only if $\log(1/\mathrm{x}) < 0$).
2. Present a cubic time algorithm to print out such a cyclic sequence if it exists.

Graph Conversion:
Let the given directed graph be G. We convert G into another graph H such that $V_H = V_G$, and $\forall\ e_G = (x, y) \in E_G$, $\exists\ e_H = (x, y) \in E_H$, such that wt($e_H$) = -log(R(x,y)), where R(x,y) = wt(e_G).

Additionally, we assume that the given graph G and consequently H are strongly connected as is expected from a practical scenario, and even if it is not, we can consider our problem as a collection of smaller sub-problems, each of which deal with different strongly connected component of the given graph.

Solution for (a):
We need to check if a cycle $(c_{i_1}, c_{i_2}, ...c_{i_k+1} = c_{i_1})$ exists in G, such that:

$\prod_{j=1}^{k} R(c_{i_j}, c_{i_j+1}) > 1 \implies$
$\log(\prod_{j=1}^{k} R(c_{i_j}, c_{i_j+1})) > 0 \implies$
$\sum_{j=1}^{k} \log(R(c_{i_j}, c_{i_j+1})) > 0 \implies$
$\sum_{j=1}^{k} wt_H(c_{i_j}, c_{i_j+1}) < 0$

So, the given problem is same as detecting a cycle $(c_{i_1}, c_{i_2}, ...c_{i_k+1} = c_{i_1})$ of negative weight in H. We will use Bellman-Ford algorithm (with parent pointers for backtracking) as discussed in the class to detect and obtain such a negative cycle, if it exists. We just provide the pseudocode for algorithm.

Note: Time complexity analysis and argument for correctness follow after the pseudo-code on next page

Algorithm 4 Pseudocode for detecing a negative cycle

dist ← an integer array of size $|V_H|$ which stores the distance value of that vertex from an arbitrary source vertex s.
parent ← array of size $|V_H|$ which stores the parent of that vertex.

```
procedure detectNegCycle(V,E):
  for all i in range(0,|V|) do
      dist[i] ← ∞
      parent[i] ← null
  end for

  dist[s] ← 0                                          ▷ s is an arbitrary source vertex
  for all i in range(1,|V|) do
      for all edges (x,y) in E do
          if dist[y] > dist[x]+wt_H(x,y) then
              parent[y] ← x
          end if
          dist[y] ← min(dist[y],dist[x]+wt_H(x,y)
      end for
  end for

  for all edges (x,y) in E do
      if dist[y] < dist[x]+wt_H(x,y) then
          return true
      end if
  end for

  return false
```

Time Complexity Analysis:
In Bellman Ford's Algorithm, there are exactly n outer iterations and exactly m inner iterations. So, the total time complexity is **O(m*n)** and if we use the fact that m = O(n^2), then **O(n^3)**, i.e. cubic in the number of vertices.

Proof of correctness:
We assume the correctness of Bellman Ford's algorithm as discussed in the class. So, starting from some arbitrary source vertex s, this algorithm finds the distance of every vertex v from s in exactly
|V| - 1 iterations, if the graph doesn't have any negative cycle. So,

Claim 3.1.1: If there is no negative cycle in G, then $\forall$ vertices v $\in V_H$, dist[v] $\leq$ dist[u] + wt_H(u,v) for some vertex u $\in V_H$ such that (u,v) $\in E_H$ after Bellman Ford's algorithm terminates.

Proof of Claim 3.1.1:
Proof by contradiction. We assume that $\exists$ some edge (u,v) $\in E_H$ such that dist[v] > dist[u] + wt_H(u,v). Then, $\exists$ a path P of length dist[u] + wt_H(u,v) from s to v which is of shorter length than dist[v]. This means that dist[v] is not the shortest distance. This contradicts the correctness of Bellman Ford's algorithm. So, the claim holds.

Note that our variant of the Bellman Ford's algorithm checks the claim 3.1.1 to detect a negative cycle. So, the algorithm **satisfactorily detects the absence of a negative cycle**. Now, it is sufficient to prove that it also detects the presence of a negative cycle.

We prove this by contradiction. Assume that the graph has a negative cycle but the algorithm doesn't detect it. In other words, the claim 3.1.1 holds even if the graph H has a negative cycle. Let the negative cycle be P = $(v_1, v_2, ...v_k, v_{k+1} = v_1)$. From our assumption,

$$\sum_{i=1}^{k} wt_H(v_i, v_{i+1}) < 0$$

Now, we know that the claim 3.1.1 holds for all vertices of V_H, so,

$$dist[v_i] \leq dist[v_{i+1}] + wt_H(e) \; \forall \; e = (v_i, v_{i+1}) \in \text{P} \implies$$
$$\sum_{i=1}^{k} dist[v_i] \leq \sum_{i=1}^{k} dist[v_{i+1}] + \sum_{i=1}^{k} wt_H(v_i, v_{i+1}) \implies$$

Since, $v_1 = v_{k+1}$, the LHS and first term of RHS are equal, and so we get:

$$\sum_{i=1}^{k} wt_H(v_i, v_{i+1}) > 0$$

which strictly contradicts our assumption. So, if there is a negative cycle in the graph, then claim 3.1.1 can never hold. So, our algorithm **correctly detects the presence of a negative cycle**. Hence Proved.

Solution for (b):
We will use the parent pointers obtained in the previous part (a) to backtrack and find a negative cycle. Note that our algorithm will correctly detect both the absence and presence of a negative cycle. So, we will only consider the case of finding the negative cycle when it exists. From part (a), we have that since a negative cycle is present in the graph, after n-1 iterations of Bellman-Ford's algorithm, $\exists$ at least one edge (u,v) $\in E_H$, such that dist[v] > dist[u] + wt(u,v). We now prove that in this case, if P denotes the new shortest path from source vertex s to v, then P contains a negative cycle.

Claim 3.2.1: After k iterations of Bellman Ford's algorithm, the length of the shortest path from s to any other vertex v is at most k.

Proof of Claim 3.2.1:
Proof this by induction on k. The claim is itself the hypothesis.

- **Base Case:** After 0 iteration, i.e. initially, dist[s] = 0 and dist[v] = ∞ for all other vertices. Now, any shortest path should contain at least 0 edges, i.e. empty paths. So, any empty path from s to s it s itself, which is of length 0. While no empty path exists between s to any other vertex v. So, dist[v] = ∞. Thus, the base case holds.
- **Induction Step:** We assume that the hypothesis holds for k i.e. after k iterations, any shortest path between s to v is of length at most k. Now, consider the $(k+1)^{th}$ iteration. There can be two types of vertices at this point:

  1. Vertex v such that dist[v] $\leq$ dist[u] + wt(u,v) $\forall$ edges (u,v) $\in E_H$. For such vertices, the shortest path doesn't change in the current iteration. And from induction hypothesis, this shortest path is of length at most k, which is less than k+1.
  2. Vertex v such that $\exists$ an edge (u,v) $\in E_H$ such that dist[v] > dist[u] + wt(u,v). Thus, the current path from s to v is not the shortest path. If the shortest path between s to u is P, then the new shortest path from s to v is P' = P $\cup$ (u,v). From induction hypothesis, the length of P is at most k, and so the length of P' is at most k+1.

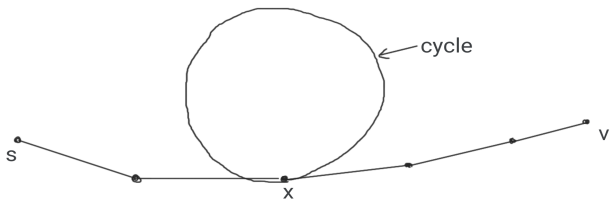
  From the above two cases, the length of shortest path from s to v is at most k+1 $\forall$ vertices v after k+1 iterations. So, the induction step holds.

From the above claim, after |V|-1 iterations of Bellman Ford's algorithm, length of shortest path from s to any vertex can not exceed (|V|-1). But for a negative cycle to exist, $\exists$ at least one edge (u,v) such that dist[v] < dist[u] + wt(u,v). So, in the |V| iterations, the edge (u,v) will be relaxed to obtain the new shortest path P from s to v.

Claim 3.2.2: The new shortest path P from s to v is exactly of length |V|.

Proof of Claim 3.2.2:
Proof by contradiction. We assume that length of path P is at most |V|-1. (Note that it can be at most |V| from claim 3.2.1). But then this shortest path could have been found in the |V|-1 iterations since from claim 3.2.1, any shortest path after k iterations can contain at most k edges. But we found the new shortest path in the $(|V|)^{th}$ iteration. We have a contradiction and so, the length of path P is exactly |V|.

From above claim, we got that the length of the new shortest path P contains exactly |V| edges. So, the number of vertices in the path is |V|+1. But there are only |V| distinct vertices. So, at least one vertex is repeated in the above path from pigeonhole principle. Let one such repeated vertex be x. So, the path is of the form P = path from s to x ∪ path from x to x ∪ path from x to v. This means that the P contains a cycle from x to x.



Note: The above figure in only not an accurate description since more than one vertices of the cycle can be a part of the path P, i.e. there may be more than one vertices which are repeated in path P. Here, we consider only one such vertex x without loss of generality.

Claim 3.2.3: This cycle C is a negative cycle.

Proof of Claim 3.2.3
Proof by contradiction. Say that this cycle C is a positive cycle. So, total length of the cycle C is positive. In such case, P' = P - C is a shorter path from s to v, which is a contradiction since we assumed that path P is the shortest path. So, C is a negative cycle.

From all the above claims, we proved that the new path from s to v contains a negative cycle. We can now obtain this path using backtracking, since the sequence of parent pointers generates the entire shortest path by tracking it in reverse direction. We use a greedy strategy to obtain the cycle.

Algorithm

1. Perform |V|-1 iterations of the Bellman Ford's algorithm and assign the distance and parent values to each of the vertices. (As in part(a))
2. Choose a vertex v such that ∃ an edge (u,v), where dist[v] > dist[u] + wt(u,v). (Existence of such vertex is already proved)
3. Assign u as the parent of v.
4. Starting from v, move to its parent and store it in a hash table. Also store the path discovered till now.
5. If any parent already appears in the hash table, it means that we have found a cycle, and we don't need to backtrack anymore. The cycle can then be obtained by traversing the discovered path in forward direction until the repeated vertex.
6. Repeat the steps 2 to 5 until such a vertex is obtained.

Algorithm 5 Pseudocode for obtaining negative cycle

dist ← an integer array of size $|V_H|$ which stores the distance value of that vertex from an arbitrary source vertex s.
parent ← array of size $|V_H|$ which stores the parent of that vertex.

```
procedure negCycle(V,E):
  for all i in range(0,|V|) do
      dist[i] ← ∞
      parent[i] ← null
  end for

  dist[s] ← 0                                          ▷ s is an arbitrary source vertex
  for all i in range(1,|V|) do
      for all edges (x,y) in E do
          if dist[y] > dist[x]+wt_H(x,y) then
              parent[y] ← x
          end if
          dist[y] ← min(dist[y],dist[x]+wt_H(x,y)
      end for
  end for

  T ← a hash table storing explored vertices
  P ← a queue storing the path discovered till now
  v ← null
  for all edges (x,y) in E do
      if dist[y] > dist[x]+wt_H(x,y) then
          parent[y] ← x
          T.insert(y)
          P.push(y)
          v ← x
          break
      end if
  end for

  while Vertex v not in T do
      T.insert(v)
      P.push(v)
      v ← parent[v]
  end while
  P.push(v)

  while P.front() not equal to v do
      P.pop()
  end while

  return P                                             ▷ This queue P is a negative cycle
```

Time Complexity Analysis:

- Bellman Ford's algorithm for the first n-1 iterations takes O(m*n) time as discussed in part (a). And the number of edges is at most O(n^2). So, total time taken in this part is **O(n^3)**.
- As discussed in the claims above, the length of the shortest path after n iterations is at most n. So, backtracking takes at most O(n) iterations. Also, each iteration is O(1), since hash table lookups are O(1) on average. So, total time taken in backtracking is **O(n)**.

So, total time complexity is **O(n^3)**.

Space Complexity Analysis:
The various data structures used in this algorithm are:

1. *dist* vector of size n
2. *parent* vector of size n
3. The hash table T, which contains at most n vertices, O(n).
4. The path discovered in backtracking can be of length at most n, so size of P is O(n).

So, total space complexity is **O(n)**.

4. Coin Change

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

1. Device a polynomial time algorithm to count the number of ways to make change for Rs. n,given an infinite amount of coins/notes of denominations, d[1],..., d[k].
2. Device a polynomial time algorithm to find a change of Rs. n using the minimum number of coins (again you can assume you have an infinite amount of each denomination).

Solution for (a):
We are given the denominations and we need to find the number of ways to make change for Rs. n using these denominations. This problem has optimal substructure so it can be solved using dynamic programming.

DP Table: Let dp[n][k] be a two dimensional integer array where n is the amount to be changed and k is the number of denominations available.

DP State: dp[i][j] represents the number of ways to obtain Rs. i using only first j denominations. ($1 \leq i \leq n$ and $1 \leq j \leq k$)

Recurrence Relation:

$$CNT(i,j) = \left\{ \begin{array}{ll} 1, & \text{if } i == 0 \\ 0, & \text{else if } j == 0 \\ CNT(i - d[j], j) + CNT(i, j-1), & \text{otherwise} \end{array} \right\}$$

Algorithm 6 Algorithm for returning the number of ways to change given amount

```
procedure CalculateWays(d[]):
  dp[][] ← 2-D array initialised to 0
  for i = 1 ; i ≤ n ; i++ do
      for j = 1 ; j ≤ k ; j++ do
          if i==0 then
              dp[i][j] = 1
          else if j==0 then
              dp[i][j] = 0
          else if i ≥ d[j] then
              dp[i][j] = dp[i-d[j][j] + dp[i][j-1]
          else
              dp[i][j] = dp[i][j-1]
          end if
      end for
  end for
  return dp[n][k]
```

Justification for correctness:

Base Case: i==0
We can show that this case is always true since if we don't have any money to exchanged, we can do this by using 0 coins which counts to one solution so dp[0][j] = 1.
j == 0
In this case, we have non zero amount to be changed but no denominations are available, so we cannot give change for this case, so number of ways will be 0 i.e. dp[i][0] = 0 $\forall i \in [1, n]$
Therefore Base case holds)

Justification for recurrence: Let us assume we need to calculate the number of ways to give change for Rs. i using first j denominations and further suppose we have calculated the number of ways to change coins for all value of Rs. a such that $a \leq i$ and also we have calculated the number of ways to give change for Rs. i using first j-1 denominations i.e. we have values of dp[a][j] $\forall a < i$ and $\forall j \in [1, k]$ and we also have values of dp[i][j-1]. To calculate dp[i][j] we can have two cases:

- **Case 1 :** j^{th} denomination is included in the change
  To calculate the number of ways for this case, we can check the number of ways to get the remaining amount after excluding the value of j^{th} denomination from the amount i.e. dp[i-d[j]][j]. Note that d[j] $\leq$ i should hold since we cannot provide change using denominations which are larger than the required amount
- **Case 2 :** j^{th} denomination is not included in the change
  In this case, since we are not including the j^{th} denomination in our change, the number of ways to change the amount i using first j denominations will remain same as the number of ways to change amount i using first j-1 denominations i.e. dp[i][j-1]

To calculate the value for dp[i][j], we should consider both cases since both may return us a valid change so we should add the value for both the cases and store the result in dp[i][j]

Time Complexity Analysis :
The final answer of the problem will be dp[n][k] so we need to fill the DP Table upto this level. The total number of values to calculated is n*k and to calculate each value, we need a constant time so The time Complexity of above algorithm will be O(n*k)

Space Complexity Analysis :
The space complexity of this algorithm will depend on the implementation. One way would be store the entire DP Table and in this case the space complexity will be proportional to the number of elements in the DP Table i.e O(n*k). But since at each step we need values from either current column or the previous column so we can only store two columns at a time. Each column has n elements So the overall space complexity of the algorithm will be O(n)

Solution for part (b):

We are given the denominations and we need to find the minimum number of coins to make change for Rs. n using these denominations. This problem has optimal substructure so this can be solved using dynamic programming.

DP Table: Let dp[n][k] be a twp dimensional integer array where n is the amount to be changed and k is the number of denominations available

DP State: dp[i][j] represents the minimum number of coins to change Rs. i using only first j denominations. ($1 \leq i \leq n$ and $1 \leq j \leq k$)

Recurrence Relation:

$$OPT(i,j) = \left\{ \begin{array}{ll} 0, & \text{if } i == 0 \\ INT\_MAX, & \text{else if } j == 0 \\ 1 + min(OPT(i - d[j], j), OPT(i, j-1)), & \text{otherwise} \end{array} \right\}$$

Algorithm 7 Algorithm for returning the denominations corresponding to minimum number of coins

```
procedure getChange(d[]):
  i ← n
  j ← k
  result ← [] dp[][] ← calculateChange(d)
  while  do
    if i - d[j] ≥ 0 and dp[i][j] == 1 + dp[i-d[j]][j]  then
      result.append(d[j])
      i ← i - d[j]
    else if j-1>=0 and dp[i][j] == 1 + dp[i][j-1] then
      j ← j - 1
    else
      break
    end if
  end while
  return result
procedure CalculateChange(d[]):
  dp[][] ← 2-D array initialised to 0
  for i = 1 ; i ≤ n ; i++ do
    for j = 1 ; j ≤ k ; j++ do
      if i==0 then
        dp[i][j] = 0
      else if j==0 then
        dp[i][j] = INT_MAX
      else if i ≥ d[j] then
        dp[i][j] = 1 + min(dp[i-d[j][j], dp[i][j-1])
      else
        dp[i][j] = 1 + dp[i][j-1]
      end if
    end for
  end for
  return dp
```

Justification for correctness:

Base Case: i==0
We can show that this case is always true since if we don't have any money to exchanged, we can do this by using 0 coins so dp[0][j] = 0.
j == 0
In this case, we have non zero amount to be changed but no denominations are available, so we cannot give change for this case, so minimum number of coints will be infinite i.e. dp[i][0] = $\forall i \in [1, n]$

Therefore Base case holds)

Justification for recurrence: Let us assume we need to calculate the minimum number of coins to give change for Rs. i using first j denominations and further suppose we have calculated the minum number of coins to change coins for all value of Rs. a such that $a \leq i$ and also we have calculated the minimum number of coins to give change for Rs. i using first j-1 denominations i.e. we have values of dp[a][j] $\forall a < i$ and $\forall j \in [1, k]$ and we also have values of dp[i][j-1]. To calculate dp[i][j] we can have two cases:

- **Case 1 :** j^{th} denomination is included in the change
  To calculate the minimum number of coins for this case, we can check the minimum number of ways to coins to give change for the remaining amount after excluding the value of j^{th} denomination from the amount i.e. dp[i-d[j]][j] and add one for the current coin so we get 1+dp[i-d[j]][j]. Note that d[j] $\leq$ i should hold since we cannot provide change using denominations which are larger than the required amount
- **Case 2 :** j^{th} denomination is not included in the change
  In this case, since we are not including the j^{th} denomination in our change, the minimum number of coins to change the amount i using first j denominations will one more than minimum number of coins to change amount i using first j-1 denominations i.e. dp[i][j-1] + 1

To calculate the value for dp[i][j], we should consider both cases since both may return us a valid change so we should take the minimum of both the cases and store the result in dp[i][j]

For returning the exact denominations, we need to traverse the DP table starting with i = n and j = k. At each cell we can have two possibilities:

- Case 1 : (i>=d[j]) and (j>0) and dp[i-d[j]][j] $\leq$ dp[i][j-1]
  In this case, we can infer that the coin of j^{th} denomination is added to the change, so we add d[j] to our list and set i $\leftarrow$ i-d[j].
- Case 2 : (j>0) or ((i>=d[j]) and dp[i][j-1] $\leq$ dp[i-d[j]][j])
  In this case, we should not include the j^{th} denomination in our change and set j $\leftarrow$ j-1.

So our Algorithm will return optimal set of coins.

Time Complexity Analysis :
The final answer of the problem will be dp[n][k] so we need to fill the DP Table upto this level. The total number of values to calculated is n*k and to calculate each value, we need a constant time so The time Complexity of above algorithm will be O(n*k)

Space Complexity Analysis :
We would need to store the entire DP Table since we want to return the exact denominations and in this case the space complexity will be proportional to the number of elements in the DP Table i.e O(n*k). But if we need to just find the minimum number of coins, we can observe that at each step we need values from either current column or the previous column so we can only store two columns at a time. Each column has n elements So the overall space complexity of the algorithm will be O(n*k)