COL351: Assignment-2

Abhinav Jain
2019CS10322
Himanshu Gaurav Singh
2019CS10358

September 26, 2021

Problem 1

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

Solution. We give a dynamic programming algorithm.
Given a list $x_1, x_2, ..x_n$ of the number of problems in n chapters, let us define $f(i, X_1, X_2)$ be a boolean function which is **true** iff the first i chapters can be partitioned into sets S_1, S_2, S_3 such that $\sum_{i\in S_1} x_i = X_1$ and $\sum_{i\in S_2} x_i = X_2$. We prove the following lemma.

Lemma 0.1. *For all $i \geq 1$, $f(i, X_1, X_2)$ is true iff one(or more) of the following holds:-*

- *$f(i-1, X_1, X_2)$ is true* ***–1***
- *$X_1 \geq x_i$ and $f(i-1, X_1 - x_i, X_2)$ is true* ***–2***
- *$X_2 \geq x_i$ and $f(i-1, X_1, X_2 - x_i)$ is true* ***–3***

where $f(0, x_1, x_2) \iff x_1 = 0$ and $x_2 = 0$

Proof. We proceed by induction.
Base case. $i = 0$. In this case, no chapter is taken in any set, X_1, X_2, X_3 can only have 0 values. Thus, it is trivially true.
Induction Assume that the lemma is true for all X_1, X_2 for $1, 2, .., i$. We will look at the value of $f(i+1, X_1, X_2)$. Consider any partition of the first $i+1$ chapters such that the first two sets have X_1 and X_2 problems. Consider the set to which the $(i+1)$-th chapter belongs. We have three exclusive and exhaustive cases.

1. $x_{i+1} \in S_1$. This is possible if $X_1 \geq x_{i+1}$. In this case, removing the $(i+1)$-th element gives sets with sizes $X_1 - x_{i+1}, X_2, X_3$ using the first i chapters. Also, the existence of a collection of sets with sizes $X_1 - x_{i+1}, X_2, X_3$ using the first i chapters implies the existence of a collection of sets with sizes X_1, X_2, X_3 using the first $i+1$ chapters. (We can add the $i+1$-th chapter to S_1).

2. $x_{i+1} \in S_2$. This is possible if $X_2 \geq x_{i+1}$. In this case, removing the $(i+1)$-th element gives sets with sizes $X_1, X_2 - x_{i+1}, X_3$ using the first i chapters. Also, the existence of a collection of sets with sizes $X_1, X_2 - x_{i+1}, X_3$ using the first i chapters implies the existence of a collection of sets with sizes X_1, X_2, X_3 using the first $i+1$ chapters. (We can add the $i+1$-th chapter to S_2).

3. $x_{i+1} \in S_3$. In this case, removing the $(i+1)$-th element gives sets with sizes $X_1, X_2, X_3 - x_{i+1}$ using the first i chapters. Also, the existence of a collection of sets with sizes $X_1, X_2, X_3 - x_{i+1}$ using the first i chapters implies the existence of a collection of sets with sizes $X_1, X_2, X_3 - x_{i+1}$ using the first $i+1$ chapters. (We can add the $i+1$-th chapter to S_3).

Thus if any of the propositions $1, 2, 3$ is true, $f(i, X_1, X_2)$ is true. Also, $f(i, X_1, X_2)$ is true implies at least one of the propositions $1, 2, 3$ is true. $\square$

The following algorithm gives the optimal partition.

Algorithm 1 $OPT - PARTITION(x)$

```
1: $DP \leftarrow OPT - PARTITION - DP(x)$
2: $sum = \sum_{i \in \{1,2,..,n\}} x_i$
3: $X_1^* \leftarrow 0$ , $X_2^* \leftarrow 0, VAL^* \leftarrow n^2$
4: for $x_1 \in 1, 2, .., n^2$ do
5:     for $x_2 \in 1, 2, .., n^2$ do
6:         if $DP[n][x_1][x_2] = true$ and $\max\{x_1, x_2, sum - x_1 - x_2\} < VAL^*$ then
7:             $VAL^* \leftarrow \max\{x_1, x_2, sum - x_1 - x_2\}$
8:             $X_1^* \leftarrow x_1, X_2^* \leftarrow x_2$
9:         end if
10:    end for
11: end for
12: Initialise empty sets $S_1, S_2, S_3$
13: $i \leftarrow n$
14: $X_1 \leftarrow X_1^*, X_2 \leftarrow X_2^*$
15: while $i > 0$ do
16:     if $DP[i-1][X_1][X_2] = true$ then
17:         $S_3 \leftarrow S3 \cup \{i\}$
18:     else
19:         if $DP[i-1][X_1 - x_i][X_2] = true$ then
20:             $S_1 \leftarrow S_1 \cup \{i\}$
21:             $X_1 \leftarrow X_1 - x_i$
22:         else
23:             if $DP[i-1][X_1][X_2 - x_i] = true$ then
24:                 $S_2 \leftarrow S_2 \cup \{i\}$
25:                 $X_2 \leftarrow X_2 - x_i$
26:             end if
27:         end if
28:     end if
29:     $i \leftarrow i - 1$
30: end while
31: return $S_1, S_2, S_3$
```

Algorithm 2 $OPT - PARTITION - DP(x)$

```
1: Initialise a $3 - D$ list $DP[][][]$ with dimension $(n, n^2, n^2)$ with entries $false$
2: $DP[0][0][0] \leftarrow true$
3: for $i \in 1, 2, ..n$ do
4:     for $X_1 \in 0, 1, 2, ..n^2$ do
5:         for $X_2 \in 0, 1, 2, ..n^2$ do
6:             if $DP[i-1][X_1][X_2] = true$ then
7:                 $DP[i][X_1][X_2] \leftarrow true$
8:             end if
9:             if $X_1 \geq x_i$ and $DP[i-1][X_1 - x_i][X_2] = true$ then
10:                $DP[i][X_1][X_2] \leftarrow true$
11:            end if
12:            if $X_2 \geq x_i$ and $DP[i-1][X_1][X_2 - x_i] = true$ then
13:                $DP[i][X_1][X_2] \leftarrow true$
14:            end if
15:        end for
16:    end for
17: end for
18: return $DP$
```

Proof of correctness. We will prove the correctness of the subroutine $OPT - PARTITION - DP$ followed by the correctness of $OPT - PARTITION$.
We will prove that after $OPT-PARTITION-DP$ terminates, $DP[i][X_1][X_2] = f(i, X_1, X_2) \forall\ i, X_1, X_2$. We proceed by induction.
Base case. $i = 0$. DP is initialised to $DP[0][x_1][x_2] = true,\ if\ (x_1, x_2) = (0, 0)\ and\ false\ otherwise.$ From the lemma proved above, this is correct. Hence, the base case is true.
Induction Step. Assume that after the i-th iteration of the outer **for** loop, $DP[j][X_1][X_2] = f(j, X_1, X_2)\ \forall\ X_1, X_2$ and $j \leq i$.
Consider the $i + 1$-th iteration. Observe that the nested **for** loops in lines **4-5** iterate over all $(X_1, X_2),\ X_1, X_2 \in \{1, 2, ..., n^2\}$. In the lines **6-13**, for each such(X_1, X_2), $DP[i+1][X_1][X_2]$ is assigned true iff one(or more) of the conditions in line **6,9,12** is true. Using the induction hypothesis we know that $DP[i][X_1][X_2] = f(i, X_1, X_2)$. Substituting into the conditions in the lines **6,9,12** gives us the same conditions as in the lemma proved above. Thus, from the lemma, we can conclude that after the $i+1$-th iteration, $DP[j][X_1][X_2] = f(j, X_1, X_2) \forall\ X_1, X_2\ and\ j \leq i+1$. Hence, the induction is complete.

We will use the following lemmas to prove the correctness of $OPT - PARTITION$.

Lemma 0.2. *After the nested **for** loops in line **4-11** terminate, VAL^* is equal to the minimum possible value of the largest number of problems alloted to a person in a partition, where the minimum is taken over all possible partitions. Also, the triple $(X_1^*, X_2^*, sum - X_1^* - X_2^*)$ is equal to the number of problems alloted to each person in some optimal partition.*

Proof. From the correctness of $OPT - PARTITION - DP$, we can infer that for all X_1, X_2, $DP[n][X_1][X_2]$ is true iff there exists a partition such that the first two persons get X_1 and X_2 problems respectively. Since there are in total at most n^2 problems, we iterate over all possible (X_1, X_2) in the **for** loop from line **4-11**.

The condition enforced in line **6** ensures that for all valid partitions of the chapters, VAL^* contains the minimum maximum number of chapters alloted to one person over all partitions and X_1^*, X_2^* are the corresponding number of chapters alloted to the first and second person respectively. $\square$

Lemma 0.3. *After any iteration of the **while** loop, $DP[i][X_1][X_2]$ is true, where i, X_1, X_2 are the values of the corresponding variables in the algorithm after some iteration is over, or initially.*

Proof. We will proceed by induction.
Base case. Initially, $X_1 = X_1^*, X_2 = X_2^*$ and $i = n$. From the **Lemma 0.2**, we can infer that $DP[n][X_1][X_2]$ is true. Hence, the base case holds.

Induction Step Assume that after the iteration with value of the **loop counter** as i, the hypothesis holds.Consider the next iteration(for which i is decremented by 1).

From the correctness of the $OPT-PARTITION-DP$ subroutine, we can infer that $DP[i][X_1][X_2] = f(i, X_1, X_2) = true$. From **Lemma 0.1**, we can conclude that one of the conditions **1,2,3** will hold. Observe that these are same as the conditional statements **16,19,23** within the **while** loop and X_1, X_2 are modified accordingly. Thus, the hypothesis will hold for the values of X_1, X_2 after the $i - 1$-th iteration too. Hence, the induction is complete. $\square$

Finally, we will prove that after the **while** loop from line **13-28** terminates, $|S_1| = X_1^*,\ |S_2| = X_2^*$.
Observe that in each iteration of the while loop, whenever we decrement X_1 by x_i we add x_i to S_1. Similar fact holds for S_2 and X_2 as well. Initially, S_1, S_2 are empty and X_1, X_2 are X_1^*, X_2^* respectively. Thus, after every iteration $\sum_{x \in S_1} = X_1^* - X_1$ and $\sum_{x \in S_2} = X_2^* - X_2$. Observe that the **while** loop terminates when $i = 0$(termination occurs always since in each iteration, i is decremented by 1).
From **Lemma 0.3**, $DP[0][X_1][X_2]$ should be true after termination. Using **Lemma 0.1** here enforces $X_1 = 0, X_2 = 0$.

Thus, $\sum_{x \in S_1} = X_1^*$ and $\sum_{x \in S_2} = X_2^*$ after the **while** loop terminates. Observe that in each iteration of the **while** loop, x_i is inserted into exactly one of the sets S_1, S_2, S_3. Thus, $\sum_{x \in S_3} = sum - X_1^* - X_2^*$. From, **Lemma 0.2**, we conclude that S_1, S_2, S_3 is an optimal partition.

Time complexity. In $OPT - PARTITION - DP$, here are 3 nested loops of ranges n, n^2, n^2. Inside them $O(1)$ operations are done,thus its time complexity is $O(n^5)$. The for loop from line 4-11 has time complexity $O(n^4)$. The while loop from line **15-30** runs n times, with $O(1)$ operations in each iteration. Thus, the total time complexity of $OPT - PARTITION$ is $O(n^5)$.

Problem 2

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

Solution. We model the problem as a directed graph with vertices as the courses and edges corresponding to the tail of the edge being the prerequisite of its head. Formally, we construct a directed graph G such that $G.V = C$ and $G.E = \{(c_1, c_2) \iff c_1 \in P(c_2) \; \forall \; c_1, c_2 \; \}$

Lemma 0.4. *If G contains a cycle, no such ordering exists. Otherwise a topologically sorted ordering of the vertices is a valid ordering.*

Proof. Observe that in any valid ordering of the courses(say $v_1, v_2, .., v_n$), for every edge of G (v_i, v_j), $i < j$ must hold. Otherwise, if there is an edge (v_i, v_j) with $i > j$, then that would imply that the course v_j has a prerequisite v_i but v_j has been taken before v_i.
Assume there is a cycle C in G. Let the vertices of C be $(v_{i_1}, v_{i_2}, ..., v_{i_k})$ where $i_1 < i_2 < ... < i_k$. Among the edges of C, consider the edge that points to v_{i_1}. This edge must be from some $j > i_1$. This implies that an edge with its head placed left to its tail in the ordering exists. Thus, a valid ordering does not exist for G.
Now, assume that G is acyclic, consider a topological ordering $(v_1, v_2, .., v_n)$. Every edges goes from left to right in the topological ordering(by definition), hence for each course, all its prerequisites are taken before it. Hence, it is a valid ordering. $\square$

Algorithm 3 $VALID\_ORDERING(C, P)$

```
1: Initialise graph $G$ with $G.V = C$ and $G.E = \{(c_1, c_2) \iff c_1 \in P(c_2) \; \forall \; c_1, c_2 \; \}$
2: Initialise empty list $L$
3: $Initialise\ list\ f\ of\ size$ n $tostorefinishtimes$
4: $Initialise\ list\ marked\ with\ n\ entries$ 0
5: $Initialise\ variable\ flag\ to$ false
6: $Initialise\ timer\ to$ 0
7: for $v \in G.V$ do
8:     if $marked_v = 0$ then
9:         $DFS(G, v, L, marked, timer, f)$
10:    end if
11: end for
12: $T \leftarrow reverse(L)$
13: for $(u, v)$ $in\ G.E$ do
14:     $(T_i, T_j) \leftarrow (u, v)$
15:     if $i > j$ then
16:         $flag \leftarrow true$
17:     end if
18: end for
19: if flag then
20:     return No valid ordering
21: else
22:     return $T$
23: end if
```

Algorithm 4 $DFS(G, v, L, marked, f)$

```
$timer \leftarrow timer + 1$
$marked_v \leftarrow 1$
for $u \in N(v)$ do
    if $marked_u = 0$ then
        $DFS(G, u, L, marked, f)$
    end if
end for
$timer \leftarrow timer + 1$
$f_v \leftarrow timer$
$L.append(v)$
```

Proof of correctness. We prove the following lemmas.

Lemma 0.5. *T is arranged in decreasing order of finish time of vertices.*

Proof. Observe that DFS is called exactly once for each vertex, thus T is indeed a permutation of the vertices of G. For any $u, v \in G.V$, suppose $f_v > f_u$, this implies v was appended to L after u because any vertex v is appended to L as soon as recursive call for it finishes. Since $T = reverse(L)$, T is sorted in decreasing order of finish time. □

Lemma 0.6. *G is cyclic $\iff$ there exists $(T_i, T_j) \in G.E$ such that $i > j$.*

Proof. Consider any cycle $C = (T_{i_1}, T_{i_2}, ..T_{i_k})$ in order. WLOG, Assume that $i_1 = min(i_1, i_2, ..i_k)$. Consider the edge in C that points to i_1. For this edge $T_{i_k} > T_{i_1}$.

For the converse, we claim that T_j is visited before T_i if there exists $(T_i, T_j) \in G.E$ such that $i > j$.
Suppose T_i is visited before T_j, since T_j is reachable from $G(\{T_i, T_j\} \in G.E)T_i$, T_j lies in the DFS-subtree of T_i implying finish time of T_i being greater than that of T_j which contradicts that T is sorted in decreasing order of finish time. Therefore, T_j is visited before T_i.

Suppose there exists $(T_i, T_j) \in G.E$ such that $i > j$. As proved above, T_j is visited before T_i which implies $DFS(T_j)$ is called before $DFS(T_i)$. Also, since T is sorted in decreasing order of finish time, $f_{T_j} > f_{T_i}$ which implies that $DFS(T_i)$ terminates before $DFS(T_j)$. This implies that $DFS(T_i)$ is called recursively by $DFS(T_j)$ and therefore, T_i is a descendant of T_j in the DFS-tree. Thus, there exists a path P form T_j to T_i. $P \cup \{T_i, T_j\}$ is a cycle in G. □

Observe that the variable flag turns true if there exists some edge (T_i, T_j) for which $i > j$. From **Lemma 0.6** and **Lemma 0.3**, this implies no valid ordering of the courses exist. Otherwise, if all edges $\{T_i, T_j\}$ have $i < j$, this implies that for each course, it prerequisites occur before it in the ordering. Therefore, it is a valid ordering. Hence, **Proved**.

Time Complexity We call DFS for each unvisited vertex. Since each vertex is visited atmost once. Total $O(n + m)$ time is taken in the depth-first-searches. The for loop from line **13-18** runs m times with $O(1)$ operations in each iteration. Hence, the total time complexity is O(n+m). Since number of edges is atmose $n * (n - 1)/2(O(n^2))$. Time complexity is $O(n^2)$.

b. 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?

Solution. We prove the following lemma.

Lemma 0.7. *The length of the largest path in G is the optimal number of semesters, where length is defined as the number of vertices in the path.*

Proof. Let the optimal number of semesters be l and the length of the largest path in G be s. Consider a longest path $P(v_1, v_2, ..v_s)$. Since for all i, v_i needs to be taken strictly before v_{i+1}, $l \geq s$. To prove that $l = s$, we will prove that a solution with number of semesters equal to s always exists.

Lemma 0.8. *In G, assign to every vertex v, the s_v-th semester where s_v is the longest simple path in G, ending at v. For this assignment, $s_u < s_v$ for all $(u, v) \in G.E$. The maximum value of s_v for this assignment will be s.*

Proof. We proceed by contradiction. Assume that $s_u \geq s_v$ for some $(u, v) \in G$. Suppose P is the longest single path ending at u, this path will have length s_u by definition. Since $(u, v) \in G$, the path $P' = P \cup (u, v)$(observe that P' is indeed a simple path, because, $v \in P$ would imply that the portion of P from v to u along with the edge (u, v) forms a cycle in G, which is assumed to be a DAG leading to a contradiction.) has length $s_u + 1$. Since, s_v is the largest path ending at v, $s_v \geq s_u + 1$, in contradiction with the initial hypothesis.
Also, the maximum value of s_v will be the maximum length of a path ending at some vertex which will be s itself(Since every simple path ends at some vertex). ☐

Thus, there exists a schedule with s semesters. ☐

Let $N(u)$ be set all vertices to which there is an edge from u. The following algorithm calculates the minimum number of semesters required.

Algorithm 5 $FAST-DEGREE-COMPLETE(C, P)$

```
$L \leftarrow VALID-ORDERING(C, P)$
*Initialise list D of length n with all entries* 1.
**for** $count \in n..1$ **do**
    $u \leftarrow L_{count}$
    **for** $v \in N(u)$ **do**
        $D_u \leftarrow max(D_u, D_v + 1)$
    **end for**
**end for**
**return** $max(D_1, D_2, .., D_n)$
```

Proof of correctness. We first prove the following lemma.

Lemma 0.9. *For a vertex u in G, assume that d_u is the length of the longest path beginning at u. Then, the following is true for all u,*

$$d_u = 1, \; if \; |N(u)| = 0$$
$$d_u = \max_{v \in N(u)} d_v + 1, \; otherwise$$

Proof. If $N(u)$ is empty, then no edge comes out of u in G, thus there is only one(trivial) path containing u only, that begins at u. Hence, $d_u = 1$.
Otherwise, consider any $v \in N(u)$. Let P_v be the longest path beginning at v. Observe that the path P_v cannot contain u, because if it contains u, then $(u, v) \cup P'(portion\ of\ the\ path\ from\ uto\ v)$ forms a cycle in G, which contradicts the fact that G is a DAG.
Since, $u \notin P_v$, $(u, v) \cup P(v)$ is a simple path beginning at v. Since P_u is the longest such path, $length(P_u) \geq length((u, v) \cup P_v)$ which implies $d_u \geq d_v + 1$, for all v.–**1**
Consider the path P_u, since $N(u)$ is non-empty, the longest path has more than one vertex. Consider the vertex v' adjacent to u in P_u. Consider the path $P' = P/\{u, v'\}$(the portion of the path P starting from v'). P' is simple path starting at v'. Since, $d_{v'}$ is the length of the longest such path, $d_{v'} \geq length(P')$. Thus, $d_u = length(P) = 1 + length(P') \leq d_{v'} + 1$. From 1, we conclude that $d_u = d_{v'} + 1$–**2**
From 1 and 2, we conclude that $d_u \geq d_v + 1$, for all v and equality is always achieved for some v. Hence, $d_u = \max_{v \in N(u)} d_v + 1$. ☐

Assume d_v to be the length of the largest simple path beginning at v in G. We prove that after the **for** loop terminates $D_v = d_v$ for all $v \in G.V$ using induction.

Proof. **Inductive Hypothesis.** After the iteration of the outer **for** loop in which **count** is assigned i, $D_j = d_{L_j}$for all $i \leq j \leq n$.
Base case $count = n$. As proved in $2(a)$, all edges in G with starting vertex as L_i have ending vertex L_j, where $j > i$. Since, L_n is the last element in L, there are no edges from L_n,thus $|N(L_n)| = 0$ and $d_{L_n} = 1$. Since $|N(L_n)| = 0$, the inner **for** loop is executed 0 times for $i = n$-th iteration. Hence, $D_n = 1 = d_{L_n}$.

Assume that the claim is true for the i-th iteration. Consider the vertex $u = L_{i-1}$ chosen in the $i-1$-th iteration. If $|N(L_u)| = 0$, the the inner **for** loops runs zero times and after this iteration is complete $D_{i-1} = 1$, same as d_{L_u}.
Otherwise, form $2(a)$, we know that all edges starting at L_{i-1} end at L_j where $j \geq i$. After the inner **for** loop is complete, D_j will be assigned the value $\max_{v \in N(u)} D_v + 1$. By the induction hypothesis $D_j = d_{L_j} \forall j \geq i$, hence

$D_u = \max_{v \in N(u)} d_{L_v} + 1$ after the $i-1$-th iteration. Using **Lemma 0.6**, $d_u = D_u$ at the end of the iteration. Hence, the induction is complete. $\square$

Thus, we conclude that at the end of the outer **for** loop, $D_i = d_{L_i} \forall i \in 1, 2, ..n$. Since any simple path begins at some vertex, $max(D_1, D_2, .., D_n)$ is the longest simple path in G. Hence, proved.

Time complexity The time complexity of $VALID - ORDERING$ is $O(n+m)$. For each value of **count**, the inner **for** loop runs as many times as there are outgoing edges from the vertex L_{count}. Thus, the total time complexity of the loop is the number of outgoing edges in G(say m). Thus, the total time complexity is $O(n+m)$. Since, m can be at most $n*(n-1)/2(O(n^2))$, the total time complexity of the algorithm is $O(n^2)$.

c. 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 a pair set $P \subset C \times C$ such that for any $(c, c') \in P$, the intersection $L(c) \cap L(c')$ is empty.
Solution. We will use a graph G' with edges reverse of G defined above. The following algorithm produces the pair set.

Algorithm 6 $PAIR - SET(C, P)$

```
1: $Initialise\ empty\ set\ P$
2: Initialise graph $G'$ with $G.V = C$ and $G.E = \{(c_2, c_1) \iff c_1 \in P(c_2)\ \forall\ c_1, c_2\ \}$
3: $Initialise\ 2-D\ list\ vis\ of\ dimension\ (n,n)\ with\ all\ entries\ false$
4: for $c_i \in G'.V$ do
5:     $DFS(c_i, vis_i)\ \#vis_i\ is\ one\ row\ of\ vis$
6: end for
7: for $c_1 \in G'.V\}$ do
8:     for $c_2 \in G'.V/\{c_1\}$ do
9:         $flag \leftarrow\ true$
10:        for $v \in G'.V/\{c_1, c_2\}$ do
11:            if $vis_{c_1}[v]\ and\ vis_{c_2}[v]$ then
12:                $flag \leftarrow false$
13:            end if
14:        end for
15:        if flag then
16:            $P.append(\{c_1, c_2\})$
17:        end if
18:    end for
19: end for
20: $return$ **P**
```

Algorithm 7 $DFS(c, G', vis)$

```
$vis_c \leftarrow true$
for $c' \in N(v)$ do
    if not $vis_{c'}$ then
        $DFS(c', G', vis)$
    end if
end for
```

Proof of correctness. We prove the following lemmas.

Lemma 0.10. *A course c' needs to be taken before taking c, iff there is a path in G' from c to c'. In other words, $L(c) = \{c' : c \neq c'$ and c' is reachable from c in $G'\}$.*

Proof. If there is a path from c to c'(say $c_0(=c), c_1, c_2, ...c_k(=c')$), this implies, c_i can be done only after c_{i+1} for all $i \in 0, 1, ..k-1$. Hence, c_0 can be done only after c_k.
To prove the converse, we will proceed by induction on a topological ordering of $G'.V$. Consider $L_1, L_2, ..L_n$ be a topological ordering of vertices of G. We first prove the base case $i = n$. Since it is the last vertex in the topological order, it has no outgoing edges, hence no prerequisites and no vertex other than itself reachable from it.

Assume that the hypothesis is true for all $c \in \ L_{i+1}, L_{i+2}, .., L_n$. Consider the vertex $u = L_i$. The courses that need to be taken before u are the neighbours of u along with $L(v)$ for all neighbours v of u.
Thus $L(u) = N(u) \cup \bigcup_{v \in N(u)} L(v)$–**1**.

From the definition of topological ordering, all neighbours L_j of u have $j \geq i$. Using the induction hypothesis, we conclude that for all $v \in N(u)$, $L(v) = \{c' : c' \neq v \ and \ c' \ is \ reachable \ from \ v \ in \ G\}$–**2**.

Also, observe that every vertex reachable from u is either neighbour of u or is reachable from a neighbour of u. Using this fact and **1** and **2**, we can conclude that $L(c) = \{c' : c \neq c' \ and \ c' \ is \ connected \ to \ u \ in \ G\}$. Hence, the induction is complete. □

Lemma 0.11. *After the execution of the subroutine $DFS(c, G', vis)$ is complete, $L(c) = \{c' : c' \neq c \ and \ vis_{c'} \ is \ true\}$.*

Proof. We know that $DFS(c, G', vis)$ will visit all the vertices in G' that are reachable from c. Thus, the set of all vertices v for which vis_v is true, is exactly the set of those vertices that are reachable from c. From Lemma 0.7, we can conclude that $L(c) = \{c' : c' \neq c \ and \ vis_{c'} \ is \ true\}$. □

Observe that the **for** loops in line **7** and **8** iterate through all $\{(c_1, c_2) : c_1, c_2 \in C, c_1 \neq c_2\}$. We will prove that (c_1, c_2), is added to P iff $L(c_1) \cap L(c_2) = \phi$.

From the lemma proved above, we can conclude that $L(c_1) = \{c' : c' \neq c_1 \ and \ vis_1[c'] \ is \ true\}$ and $L(c_2) = \{c' : c' \neq c_2 \ and \ vis_2[c'] \ is \ true\}$.

Consider the variable **flag** in the algorithm. Observe that after the **for** loop in line **10** terminates, **flag** is assigned false iff there exists $v \in G.V/\{c_1, c_2\}$ such that $vis_1[v]$ and $vis_2[v]$ are both. This occurs iff the sets $\{v : v \neq c_1 \ and \ vis_1[v] = true\ \}$ and $\{v : v \neq c_2 \ and \ vis_2[v] = true\ \}$ have atleast one element in common.

From **Lemma 0.8**, we conclude that **flag** is assigned false iff $L(c_1) \cap L(c_2) \neq \phi$. Thus in line **16**, (c_1, c_2) is added to P iff **flag** is true or $L(c_1) \cap L(c_2) = \phi$. Hence, **Proved**.

Time complexity. The **for** loop at line **10** runs $n - 2$ times. Lines **11** and **12** are constant time operations, hence total time complexity of this loop is $O(n)$. The nested **for** loops in line **7** and **8** run a total of $n * (n-1)(O(n^2))$ times. Thus total time complexity for the lines **7-19** is $O(n^3)$ –**1**. For the lines **4-6**, we do n DFS calls, each taking $O(n+m)$ time. Thus, time complexity for lines **4-6** is $O(n * (n+m))$, since there can be at most $n * (n-1)/2$ edges in G, m is $O(n^2)$, thus complexity is $O(n^3)$.–**2**. From **1** and **2**, we can conclude that total time complexity is $O(n^3)$.

Problem 3

(a) Suppose you are a trader aiming to make money by taking advantage of price differences between different currencies. You model the currency exchange rates as a weighted network, wherein, the nodes correspond to n currencies - $c_1, ..., c_n$, and the edge weights correspond to exchange rates between these currencies. In particular, for a pair (i, j), the weight of edge (i, j), say R(i, j), corresponds to total units of currency c_j received on selling 1 unit of currency c_i Design an algorithm to verify whether or not there exists a cycle $(c_{i1}, ..., c_{i_k}, c_{i_{k+1}} = c_{i_1})$ such that exchanging money over this cycle results in positive gain, or equivalently, the product $R[i_1, i_2].R[i_2, i_3]...R[i_{k-1}, i_k]R[i_k, i_1]$ is larger than 1.

Solution We observe that the inequality

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

is equivalent to

$$log(R[i_1, i_2].R[i_2, i_3]...R[i_{k-1}, i_k]R[i_k, i_1]) > 0$$

which is equivalent to

$$log(R[i_1, i_2]) + log(R[i_2, i_3]) + ... + log(R[i_{k-1}, i_k]) + log(R[i_k, i_1]) > 0$$

which is equivalent to

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

Therefore, if we consider a weighted directed graph G with vertices $\{c_1, c_2, .., c_n\}$ such that for each pair (i, j), the weight of edge (c_i, c_j), is $-log(R(i, j))$, then the problem reduces to detecting a negative weight cycle in this graph. We will use the **Bellman Ford algorithm** discussed in the class to detect a negative cycle in a graph.

Algorithm 8 *Currency Exchange*(n, c)

```
1: Create a 1D array $d$ of size $(n + 1)$.
2: Initialise $d[1] = 0$ and $d[i] = \infty$ for each $i = 2, ..., n$
3: Initialise $parent[0] = null$ for each $i = 1, ..., n$
4:
5: for $r = 1, 2, ..., n - 1$ do
6:     for $i = 1, 2, ..., n$ do
7:         for $j = 1, 2, ..., n$ do
8:             if $d[j] > d[i] + (-log(c_j/c_i))$ then
9:                 $d[j] = d[i] + (-log(c_j/c_i))$
10:                $parent[j] = i$
11:            end if
12:        end for
13:    end for
14: end for
15:
16: for $i = 1, 2, ..., n$ do
17:     for $j = 1, 2, ..., n$ do
18:         if $d[j] > d[i] + (-log(c_j/c_i))$ then
19:             Print "Cycle found"
20:         end if
21:     end for
22: end for
23: Print "No Cycle found"
```

Proof of correctness. The **Bellman Ford** algorithm is used to detect a negative-weight cycle that is reachable from the start vertex. Since an edge with weight $R[i, j]$ exists between all vertices i, j, the graph is strongly connected and hence every cycle will be reachable from the start vertex. Thus, the algorithm is able to find the existence of any negative-weight cycle in the graph.

Time complexity Observe that O(1) amount of computation is done in the innermost loop and the total loop iteration is O(n^3). Therefore, the time complexity is O(n^3).

(b) Present a cubic time algorithm to print out such a cyclic sequence if it exists.

Solution We will again use the **Bellman Ford** algorithm to print the negative cycle if there exists one.

Algorithm 9 Currency Exchange(n,c)

```
Create a 1D array $d$ of size $(n+1)$.
Initialise $d[1] = 0$ and $d[i] = \infty$ for each $i = 2, ..., n$
Initialise $parent[0] = null$ for each $i = 1, ..., n$
for $r = 1, 2, ..., n-1$ do
    for $i = 1, 2, ..., n$ do
        for $j = 1, 2, ..., n$ do
            if $d[j] > d[i] + (-log(c_j/c_i)$ then
                $d[j] = d[i] + (-log(c_j/c_i)$
                $parent[j] = i$
            end if
        end for
    end for
end for
$v \leftarrow -1$
for $i = 1, 2, ..., n$ do
    for $j = 1, 2, ..., n$ do
        if $d[j] > d[i] + (-log(c_j/c_i)$ then
            $parent[j] = i$
            $v \leftarrow j$
            break
        end if
    end for
end for
if $v = -1$ then
    Print "No Cycle found"
else
    Print "Cycle found"
    for $i = 1, 2, ..., n$ do
        $v \leftarrow parent[v]$
    end for
    $x, cycle \leftarrow v, \{\}$
    do
      $cycle.append(v)$
      $v \leftarrow parent[v]$
    While($v \neq x$)
    reverse($cycle$)
    for $u \in cycle$ do
        Print $u$
    end for
end if
```

Proof of correctness : First we observe that there is an edge between any two nodes in the graph. Hence the graph is strongly connected and if there is a negative cycle in the graph, then it is reachable from the source vertex (which is vertex 1 in our case) used in **Bellman Ford** algorithm.

First we prove that if a graph doesn't contain a negative weight cycle, then the **Bellman Ford** algorithm returns "cycle not found". Since all the cycles in the graph have a non=negative weight, to each vertex there exists a shortest simple path from source (if there is a cycle in the path we can remove the cycle from the path getting a smaller path weight). Consider a shortest simple path $P = v_0, v_1, ..., v_k$ where v_0 is the source vertex (say s). Observe that $k < n$ i.e there are at most n-1 edges in the shortest path as the path doesn't contain any cycle. Now we will give a proof by induction on number of iterations of outer **for** loop:

Base Case ($r = 0$): Since we have initialised $d[s] = 0$, the base case holds.

Induction hypothesis : Shortest distance has been computed correctly for all vertices which have a shortest path of length less then k.

Induction step ($r = k$) : Consider a vertex v_k which has a shortest path of length k say $P = v_0, v_1, ..., v_k$. By induction hypothesis $d[v_{k-1}]$ has been computed correctly and now in the k^{th} iteration when we pass through the edge (v_{k-1}, v_k), the value of $d[v_k]$ will be updated and it will be equal to weight of shorted path P.

Since the shortest path contains at most $n - 1$ edges, hence after $n - 1$ iterations of outer loop $d[v_i]$ will contain the correct value of shortest path for all vertices v_i reachable from source vertex. Hence in the n^{th} iteration, no distances will be updated and the algorithm will return "cycle not found".

Now we prove that if a graph contains a negative weight cycle than the bellman ford algorithm returns "cycle found". Consider a negative weight cycle $C = v_0, v_1, ..., v_k$ where $v_0 = v_k$ and $W(v_0, v_1) + W(v_1, v_2) + ... + W(v_{k-1}, v_k) < 0$. Assume for the sake of contradiction that the algorithm returns "cycle not found". Then we have $d[v_i] \leq d[v_{i-1}] + W(v_{i-1}, v_i)$ for each vertex on the cycle i.e $i = 1, ..k$. Summing up all these inequalities and using the fact that $v_0 = v_k$ we obtain $0 \leq W(v_0, v_1) + W(v_1, v_2) + ... + W(v_{k-1}, v_k)$, which is a contradiction. Hence if there is a negative cycle in the graph then the bellman ford algorithm correctly returns "cycle found".

Now suppose there is a vertex v for which the value of $d[v]$ changes (say to $d'[v]$) in the n^{th} iteration of the bellman ford algorithm. Observe that the path of cost $d'[v]$ (say P) from source to v must use exactly n edges, hence it must contain a cycle. Now observe that if the cycle had a non-negative weight than the value of $d[v]$ would not have changed in the n^{th} iteration as removing the cycle from the path does not increase the cost of path and it would contain lesser number of edges. Hence, the cycle has negative weight. In order to find the cycle, we iterate through the path P by successively jumping n times through the parent of v, reaching the vertex u. Observe that u must lie on the cycle as there are at most n vertices in the path P. Now, to find all vertices on the cycle we simply trace through the parent of u until we reach u.

Time complexity When calculating the value of shortest distance, observe that $O(1)$ amount of computation is done in the innermost loop and the total loop iterations are $O(n^3)$. If a cycle if found than $O(n)$ steps are performed in printing the cycle. Hence the total time complexity is $O(n^3)$.

Problem 4

(a) 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, $d[1], ..., d[k]$.

Solution First we note that in counting the number of ways to make change, the ordering of coins is not considered. Thus, we are going to count the number of distinct multisets $\{c_1, c_2, ..., c_m\}$ such that $d[c_1] + d[c_2] + .. + d[c_m] = n$. Let $T(i, w)$ denote the number of such multisets to make change of Rs. w using the coins of the first i denominations only. Consider any valid collection($\{c_1, c_2, ..., c_m\}$).WLOG, assume $c_1 \leq c_2 \leq .. \leq c_m$. There are two possible exhaustive cases :
1.) $c_m = i$. In this case, the rest of the coins belong to $\{1, 2, .., i\}$ and there sum must be $w - d[i]$, therefore the number of possible such multisets are $T(i, w - d[i])$, provided that $w \geq d[i]$.
2.) $c_m < i$. In this case, all the coins belong to $\{1, 2, ..i - 1\}$ with sum being w. Therefore, the number of possible such multisets are $T(i - 1, w)$.

Combining the 2 cases, we obtain the following recurrence relation :

$$T(i, w) = T(i, w - d(i))\{w \geq d[i]\} + T(i - 1, w) - \mathbf{1}$$

where $w \geq d[i]$ is an indicator variable which is 1 when $w \geq d[i]$ and 0 otherwise. Further we note that $T(i, 0) = 1$ for all $i \geq 0$ and $T(0, w) = 0$ for all $w \geq 1$. Now we will give a dynamic programming algorithm to count the number of ways to make change :

Algorithm 10 *Change*(d, k, n)

```
Create a 2D array $dp$ of size $(k + 1) * (n + 1)$.
Initialise $dp[i, 0] = 1$ for each $i = 0, ..., k$
$dp[0, w] = 0$ for each $w = 1, ..., n$
for $i = 1, 2, ..., k$ do
    for $w = 1, 2, ..., n$ do
        if $w \geq d[i]$ then
            $dp[i, w] = dp[i, w - d(i)] + dp[i - 1, w]$
        else
            $dp[i, w] = dp[i - 1, w]$
        end if
    end for
end for
return $dp[k][n]$
```

Proof We will give the proof of correctness by an inductive argument.

Base case $(i = 0)$: Base case holds as we have initialised $dp[0, 0] = 1$ and $dp[0, w] = 0$ for $w \geq 1$.

Induction hypothesis : Assume that the value of $dp[i, w]$ computed by the algorithm is equal to the number of ways to make change of Rs. w using the coins of the first r denominations.

Induction step $(i = r + 1)$:
Now we will do induction on w to prove that $dp[r + 1][w]$ holds for all $w = 0, 1, ...n$

Base case $(w = 0)$: Base case holds as we have initialised $dp[r + 1, 0] = 0$.
Induction hypothesis : Assume that the value of $dp[r + 1, w]$ computed by the algorithm is equal to the number of ways to make change of Rs. w using the coins of the first $r + 1$ denominations for all $w \leq q$.
Induction step $(w = q + 1)$: Observe that the computation done in the inner loop is equivalent to $dp[r + 1, q + 1] = dp[r+1, q+1-d[r+1]]\{w \geq d[r+1]\}+dp[r, q+1]$. By inductive hypothesis $dp[r, q+1]$ and $dp[r+1, q+1-d[r+1]]$ have been computed correctly as both the outer and the inner **for** loop iterates in **increasing** order, hence $dp[r + 1, q + 1]$ is also computed computed correctly i.e it satisfies the above stated recurrence relation (1)
Hence by PMI value of $dp[r+1, w]$ is computed correctly for all w = 0,1,...n. The induction step for $i = r+1$ is complete.

Hence by PMI value of $dp[i, w]$ is computed correctly for all $i = 0, 1, .., k$ and $w = 0, 1, ...n$.

Time complexity : For each iteration of outer loop there are n iteration of inner loop and in each step of inner loop O(1) steps are done. There are k iterations of outer loop. Therefor the total time complexity is O(nk).

Space complexity : The dp array takes $O(nk)$ space.

(b) 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 We will use dynamic programming to solve the problem. Let $T(n)$ denote the minimum number of coins required to find a change of Rs. n. Then the following recurrence relation holds.

$$T(0) = 0$$
$$T(n) = \infty \text{ if } n < 0$$
$$T(n) = \min_{i=1,2..k} T(n - d[i]) + 1 \text{ ,otherwise}$$

Proof. Observe that there is no way to make negative change hence we have set $T(n)$ to be infinity for negative n and for making change for Rs. 0 we require 0 coins. Hence $T(0)$ is 0.
Now to make a change of positive amount n we need to pick atleast 1 coin. Suppose that the coin is of denomination $d[i]$. Then we have to make the remaining change $(n - d[i])$ and the optimal way to do it would require $T(n - d[i])$ coins. Hence optimal way to make change of amount n containing a coin of denomination $d[i]$ is $T(n-d[i])+1$. Hence we can say that $T(n) \leq T(n - d[i]) + 1\ \forall i$, such that $n \geq d[i]$. **- 1**.
Let $d[i']$ be the denomination of a coin in the optimal solution S' of size $T(n)$ to make change of Rs.n. This implies $n \geq d[i']$.
Removing this coin from S' gives a way to make change of $n - d[i']$ using $T(n) - 1$ coins. Thus, we can we say $T(n - d[i']) \leq T(n) - 1$, or $T(n) \geq T(n - d[i']) + 1$.
From **1**, we conclude that $T(n) = T(n - d[i']) + 1$ **- 2**. Thus, equality is always achieved for some i in **1**.

From **1** and **2**, we conclude that $T(n) = \min_{i=1,2..k} T(n - d[i]) + 1$ holds for all positive number n. □

Now we will give a dynamic programming algorithm to calculate the minimum coins required to make a change :

Algorithm 11 Change(n)

```
Create a 1D array $dp$ of size $(n + 1)$.
Initialise $dp[0] = 0$ and $dp[i] = \infty$ for each $i = 1, ..., n$
Initialise $p[0] = 0$ and $p[i] = -1$ for each $i = 1, ..., n$
for $w = 1, 2, ..., n$ do
    for $i = 1, 2, ..., k$ do
        if $dp[w] > dp[w - d[i]] + 1$ then
            $dp[w] = dp[w - d[i]] + 1$
            $p[w] = i$
        end if
    end for
end for
$i, ls \leftarrow n, \{\}$
while $i > 0$ and $p[i] \neq -1$ do
    ls.append($d[p[i]]$)
    $i \leftarrow i - d[p[i]]$
end while

return $ls$
```

Proof We will give the proof of correctness by an inductive argument.

Base case $(w = 0)$: Base case holds as we have initialised $dp[0] = 0$ and $p[0] = 0$ as we don't require any coin to make change of Rs. 0.

Induction hypothesis : Assume that the value of $dp[i]$ computed by the algorithm is equal to the optimal way to make change of Rs. w for all $w \leq r$ and $d[p[i]]$ denotes the denomination of a coin which is contained in an optimal assignment to make change for Rs. w .

Induction step $(w = r+1)$: Observe that the computation done in the loop is equivalent to $dp[r+1] = \min_{i=1,2..k} dp[r+1-d[i]]+1$. By inductive hypothesis $dp[r+1-d[i]$ have been computed correctly for all $i = 1, 2.., k$ as loop iterates through i in **increasing** order hence $dp[r+1]$ is also computed computed correctly i.e it satisfies the above stated recurrence relation. Further p[r+1] stores the value of i for which the minimum is observed, if there is no way to generate a change of Rs. $r+1$ then both $p[r+1]$ and $dp[r+1]$ remain -1.
Hence by PMI value of $dp[r+1]$ is computed correctly for all $w = 0, 1, ...n$.
Once the values of $p[w]$ are computed for all $w = 0, 1, ...n$, the while computes the list of coins which achieve the optimal value for making the change. Note that the while loop terminates as the value of i decrements by atleast 1 in each iteration.

Time complexity : For each iteration of outer loop there are k iteration of inner loop and in each step of inner loop $O(1)$ steps are done. There are n iterations of outer loop. The while loop takes $O(n)$ steps as in each iteration of while loop the value of i decrements by atleast 1. Therefor the total time complexity is O(kn).

Space complexity : The dp and p array both take space of $O(n)$.