COL351 Fall 2021 Assignment 2

Mustafa Chasmai, Tamajit Banerjee

TOTAL POINTS

54.5 / 60

QUESTION 1

1 Algorithm Design book 15 / 15

+ **4 pts** Recurrence Relation + answer formulation(4)
+ **3 pts** Correctness argument(3)
+ **2 pts** Boundary cases(2)
+ **5 pts** Algorithm(5)
+ **1 pts** Time complexity ananlysis(1)

+ 15 Point adjustment

QUESTION 2

2 Course Planner 14 / 15

Part 1

+ **0 pts** Not attempted

✓ **+ 2.5 pts Part 1 : Correct and detailed algorithm**

+ **1.25 pts** Part 1 : Partially correct algorithm or details not clear

✓ **+ 1.5 pts Part 1 : Correct proof**

+ **0.75 pts** Part 1 : Partial or incomplete proof
+ **1 pts** Part 1 : Complexity analysis

Part 2

+ **0 pts** Not attempted

✓ **+ 2.5 pts Part 2 : Correct and detailed algorithm**

+ **1.25 pts** Part 2 : Partially correct algorithm or details not clear

✓ **+ 1.5 pts Part 2 : Correct proof**

+ **0.75 pts** Part 2: Partial or incomplete proof
+ **1 pts** Part 2 : Complexity analysis

Part 3

+ **0 pts** Not attempted

✓ **+ 2.5 pts Part 3 : Correct and detailed algorithm**

+ **1.25 pts** Part 3 : Partially correct algorithm or details not clear

✓ **+ 1.5 pts Part 3 : Correct proof**

+ **0.75 pts** Part 3 : Partial or incomplete proof
+ **1 pts** Part 3 : Complexity analysis
+ **0 pts** Not attempted

+ 2 Point adjustment

m and n in complexity analysis undefined

QUESTION 3

3 Forex Trading 14.5 / 15

✓ **- 1 pts Define the graph, vertices and edge details, and also the number and weights of the edges.(2)**

+ 0.5 Point adjustment

QUESTION 4

4 Coin Change 11 / 15

Part a

✓ **+ 3 pts Correct and detailed algorithm**

+ **2 pts** Correct but not a detailed algorithm/Minor mistakes in the algorithm

✓ **+ 1 pts Complexity analysis**

✓ **+ 3 pts Correct Proof of Correctness of the algorithm**

+ **2 pts** Minor mistakes in the Proof of Correctness of the algorithm/Not a detailed proof
+ **0 pts** Incorrect algorithm

Part b

+ **3 pts** Correct and detailed algorithm

✓ **+ 2 pts Correct but not a detailed algorithm/Minor mistakes in the algorithm**

+ **1 pts** Complexity analysis
+ **4 pts** Correct Proof of Correctness

✓ **+ 3 pts Minor mistakes in the Proof of Correctness of the algorithm/Not a detailed proof**

+ **0 pts** Incorrect algorithm

- 1 Point adjustment

COL351 Assignment 2

Tamajit Banerjee | Mustafa Chasmai
2019CS10408 | 2019CS10341

September 2021

1. **Algorithms Design Book**

   (a) Recursion (Proof of correctness) :

   Let Partition (i,j,k) be True if and only if we can partition the first k chapters { x1,x2,x3,...,xi } into 3 sets s1,s2,s3 with the sum of the chapters in s1 is j , that in s2 is k and that in s3 is $\sum_{l=1}^{i}$ x_l - j - k.

   We know, every chapter x_i has 3 probabilities. The possibilities are :

   i. x_i can belong to set s1
   ii. x_i can belong to set s2
   iii. x_i can belong to set s3

   The above cases are trivially exhaustive.

   The base cases are:

   i. Using 0 chapters , the sum of set s1 can only be 0 , that of set s2 can only be 0 and s3 can only be 0. So , Partition(0,0,0) = True and Partition(0,j,k) = False $\forall(j > 0|k > 0)$

   The other boundary cases are using any number of chapters, the sum in any set cannot be negative as a result Partition(i,j,k) is False $\forall(j < 0|k < 0)$

   The final recursion is as follows :

   $$Par(i,j,k) = \begin{cases} True, & if \text{i=0 and j=0 and k=0.} \\ False, & if \text{i=0 and j>0 and k>0.} \\ Par(i-1,j,k) or Par(i-1,j-x_i,k) or Par(i-1,j,k-x_i) & otherwise \end{cases}$$

   Proof of Correctness of Algorithm:

   Let a hypothesis H(i,j,k) is that Partition(i,j,k) is true.

   Proof by strong induction :

   The Base cases are correct as explained above.

   Induction Hypothesis : Let H(a,b,c) is true $\forall$ (a,b,c) such that (a <= i, b <= j, c <= k and (a,b,c) != (i,j,k))

   Induction step :

   Since we use the recursion relation defined above and the recursion relation is correct and the value Partition(i,j,k) only depends on Partition(a,b,c) such that (a <= i, b <= j, c <= k. and (a,b,c) != (i,j,k)).

   Hence H(i,j,k) is true from induction hypothesis.

   Hence proved

   Time Complexity:

   O(n^5)

   Line 1: We initialise a boolean array of size $(n+1) * (n^2+1) * (n^2+1)$ with all of the elements initialised to False. O(n^5)

Line 5-13: We have 3 nested for loops. The 1st loop runs for n times and each of the inner loop runs for n^2 times. Hence total time spent is O(n^5).

Line 18-24: We find the best answer possible in $O(n^4)$ time

Line 28-34: We find the partition in O(n) time.

Space Complexity:

$O(n^5)$

We use a boolean array of size $(n+1) * (n^2+1) * (n^2+1)$.

Algorithm 1: Algorithms Design Book

```
Partition ← boolean array of size (n + 1) ∗ (n^2 + 1) ∗ (n^2 + 1) with all of the elements initialised to False;
X ← { x1 , x2 , x3 , ... , xn } ;
Sum ← integer array of size (n+1) with all the elements initialised to 0 Partition[0][0][0] ← True ;
for i ← 1 to n do
    for j← 0 to n^2 do
        for k← 0 to n^2 do
            Partition[i][j][k] ← Partition[i-1][j][k] ;
            if j ≥ X[i] then
                Partition[i][j][k] ← Partition[i][j][k] or Partition[i-1][j-X[i]][k];
            end
            if k geq X[i] then
                Partition[i][j][k] ← Partition[i][j][k] or Partition[i-1][j][k-X[i]];
            end
            Sum[i] = Sum[i-1] + X[i];
        end
    end
end
answer ← inf ;
bestj ← -1 ;
bestk ← -1 ;
chapter ← n ;
for j← 0 to n^2 do
    for k← 0 to n^2 do
        if Partition[n][j][k] = True then
            if ans > max(j,max(k,Sum[n]-j-k) then
                ans ← max(j,max(k,Sum[n]-j-k) ;
                bestj ← j ;
                bestk ← k ;
            end
        end
    end
end
S1 ← null ;
S2 ← null ;
S3 ← null ;
while chapter > 0 do
    if Partition[chapter-1][bestj][bestk] = True then
        S3.append(chapter);
    else
        if bestj ≥ X[chapter]andPartition[chapter − 1][bestj − X[chapter]][bestk] = True then
            S1.append(chapter) ;
            bestj ← bestj - X[chapter] ;
        else
            S2.append(chapter) ;
            bestk ← bestk - X[chapter] ;
        end
    end
end
return S1,S2,S3;
```

1 Algorithm Design book **15 / 15**

+ **4 pts** Recurrence Relation + answer formulation(4)
+ **3 pts** Correctness argument(3)
+ **2 pts** Boundary cases(2)
+ **5 pts** Algorithm(5)
+ **1 pts** Time complexity ananlysis(1)

+ 15 Point adjustment

2. **Course Planner**

(a) Consider representing the problem in the form of a directed graph. An edge (x,y) represents the relation that the course x is a pre-requisite for course y.

Claim 1: G will have be a DAG for a valid ordering to exist.

Proof. Proof by Contradiction:
Let G not be a DAG, i.e. let it have a cycle C.
Now let v be the vertex in C that corresponds to the course that is done first. Now since v is part of a cycle, there has to be atleast one vertex u, such that (u , v) $\in$ E. But by definition, this means that u is a pre-requisite of v. Thus, v cannot be corresponding to the course that is done first. Thus, this is a contradiction. □

A valid order of courses is such that any course that is done must have all of its pre-requisites already done. Thus, the order must have the property:

$$\text{For any edge } x \to y, order(x) < order(y)$$

This is exactly the property of a topological sort.
Thus, we devise an algorithm as follows:

i. Check if the graph is a DAG or not.
ii. If the graph is not a DAG, no ordering is possible.
iii. If the graph is a DAG, perform topological sort and return the sorted order.

Algorithm 2: Course planner A

```
if not isDAG(G) then
 | No Ordering Exists!
end
O_s ← Initialised empty Stack
visited ← Initialised Array of size |V|
Function DFS_Stack(G, v):
 | Visited ← Visited + v
 | for u in neighbours(v) do
 | | if u not in visited then
 | | | DFS_Stack(G, u)
 | | end
 | | O_s.push(v)
 | end
 | return
DFS_Stack(G, V[0])
O ← [ ]
while O_s is not empty do
 | O ← O + O_s.pop()
end
```

Proof of correctness:
for any edge $u \to v$, v will be accessed either before the DFS_Stack() is called at u, in which case v is added to the stack before u, or v will be accessed in the Call of DFS_Stack(u), in which case again, v will be added to the stack before u. Since the stack is later reversed, the invariant property will be satisfied
Time Complexity: O(m+n) (DFS on DAG)

(b) Let us build a graph G where the courses are C and the edges are from the prerequisite courses of a course to that course.

Proof of Corretness:

Let us define a partitioning Par = { g_0 ,g_1 .... g_k } where g_i is the group of the courses done in the i^{th} semester.

This partition is valid if and only if the prerequisite conditions are satisfied which states that all prerequisites must be completed before taking the course.

Let Par0 = { g_1 ,g_2 .... g_k } and G0 = G \ g_0.

It is trivial that taking all the courses which have no prerequisite in the first group is a valid 1st group.

Let S be the group of all courses which have no prerequisites.

Claim 1: There exists an optimal partition Par = { g_0 ,g_1 .... g_k } such that g_0 = S

Proof. Let A be an optimal partition such that it's 1st group != g_0. Therefore, there exists a course c $\in$ g_0 that is not present in the 1st group.

We can take that course from its corresponding group and place it in the 1st group without any violation of prerequisite.

We can do it all such courses that are not present and place them in the 1st group and we will still get an optimal solution.

Hence, proved.

□

Claim 2: OPT(G) = OPT(G0) + 1

Proof. The proof is divided into two parts as follows:

i. 1+ OPT(G0) $\leq$ OPT(G)
   Let Par0 = { g_1 ,g_2 .... g_k } be an optimal partition of G \ S which is G0 (courses having group S removed) .
   Then since all courses in S have no prerequisites, Par = { g_0 ,g_1 .... g_k } is a valid partition of G.
   Since the optimal partition of the original graph cannot have more semesters than any of its solutions , we have OPT(G) $\geq$ |Par| = 1+ OPT(G0)

ii. 1 + OPT(G0) $\geq$ OPT(G)
   There exits an optimal solution Par = { g_0 ,g_1 .... g_k } such that g_0 = S. This can be derived from Claim 2.
   Par0 = { g_1 ,g_2 .... g_k } is a valid partition for G \ S which is G0.
   Since the optimal partition of C \ S cannot have more semesters than any of its solutions , OPT(G0) $\geq$ OPT(G) - 1

Hence proved. □

Time Complexity: O(m+n)

Finding degree is O(m) just iterate over all edges

We iterate over a edge at most 2 times and each vertex one time .

Hence the total complexity is O(m+n)

Algorithm 3: Course planner B

```
deg[v] ← degree of v ∀v ∈ V
S ← empty array
for v ∈ V(G) do
    if deg[v] = 0 then
        S ← S + v
    end
end
length ← 0
while S is not empty do
    S' ← empty array
    for v in S do
        for u ∈ neighbour(v) do
            deg[u] ← deg[u] − 1
            if deg[u] = 0 then
                S' ← S' + u
            end
        end
    end
    S ← S'
    length ← length + 1
end
```

(c) Proof of correctness:

Let us define L(c) as the list of all the courses that must be completed before crediting c.

We define a graph G' where the courses are the vertices and the edges are the from a course to its prerequisite course.

We claim that a course must be completed before crediting c if and only if it is reachable from C in G'.

Proof. **Part 1 :** a course must be completed before crediting c implies that it is reachable from C in G'.

We will prove the equivalent statement: If a course is not reachable from C in G' implies that it does not belong to L(c)

Construct a topological sort of the courses in graph G' with increasing order of departure time (Proof and algorithm in part1).

Iterate from i equal to 0 to |V|-1

Let us define H(i) :If a course is not reachable from c_i in G' implies that it does not belong to L(c_i)

Base case :: The one with the lowest departure time that is C0 will not have any edges going out of it so the number of reachable nodes is 1. So H(0) is trivially true

Induction Hypothesis : Let H(j) be true for all j < i

Induction :: The edges of c_i from c_i are to c_l where l < i which is true due to topological sorting.

We know, the nodes reachable from c_i are the prerequisites of c_i and union of the nodes reachable from each of its prerequisites.

So the nodes not reachable from c_i is intersection of NOT(the prerequisites) and (the nodes reachable from each of its predecessors)

Algorithm 4: Course planner C

```
1  G' = G with edges reversed ;
2  Dep ← empty list ;
3  for v ∈ V do
4  |  L ← { } ;
5  |  visited ← boolean array of size |V| initialised to false;
6  |  visited[v] ← true ;
7  |  DFS(v,visited);
8  |  for ver ∈ V do
9  |  |  if visited[ver] = true and v != ver then
10 |  |  |  L.append(ver);
   |  |  end
   |  end
11 |  Dep.append(L);
   end
12 Ans ← empty list
13 for i ← 1 to |V| do
14 |  for j ← i+1 to |V| do
15 |  |  indexi ← 0;
16 |  |  indexj ← 0;
17 |  |  temp ← false ;
18 |  |  while indexi < Dep[i].size() and indexj < Dep[j].size() do
19 |  |  |  if Dep[i][indexi = Dep[j][indexj] then
20 |  |  |  |  temp = true;
21 |  |  |  |  break ;
   |  |  |  else
22 |  |  |  |  if Dep[i][indexi] < Dep[j][indexj] then
23 |  |  |  |  |  indexi ← indexi + 1;
   |  |  |  |  else
24 |  |  |  |  |  indexj ← indexj + 1;
   |  |  |  |  end
   |  |  |  end
   |  |  end
25 |  |  if temp = true then
   |  |  |  Ans.append({i,j})
   |  |  end
   |  end
   end
26 return Ans ;
```

So the nodes not reachable from c_i is a subset of nodes not reachable from a prerequisite of c_i.

Hence by induction hypothesis , all the predecessors of c_i can be completed before c_i by using completing the nodes reachable from c_i.

Therefore we will be able to credit c_i as all its prerequisites are completed.

Hence proved Part 1

Part 2: If a course is reachable from C then it must be completed before C.

If a node v is reachable it implies there will be a path from C to that node.

Let path P be c_1 c_2 c_3 ... c_n v
Since there is an edge between each of the consecutive vertices from c_i to c_{i+1}
This implies the following :
c_2 needs to be completed before c_1
c_3 needs to be completed before c_2
....
c_n needs to be completed before c_{n-1}
v needs to be completed before c_n
All of them implies that v needs to be completed before c_1
Hence proved
Time Complexity: $O(n^3)$
We do DFS from each vertex which is O(m) for each vertex. It is O(m*n) in all.
Then we iterate over every pair and check if they have an intersection or not.
For every pair it is done in O(n) and there are $O(n^2)$ pairs.
So the total complexity is $O(m*n + n^3)$ but as m is $O(n^2)$. We can write it as $O(n^3)$

□

2 Course Planner 14 / 15

Part 1

+ **0 pts** Not attempted

✓ **+ 2.5 pts** **Part 1 : Correct and detailed algorithm**

+ **1.25 pts** Part 1 : Partially correct algorithm or details not clear

✓ **+ 1.5 pts** **Part 1 : Correct proof**

+ **0.75 pts** Part 1 : Partial or incomplete proof

+ **1 pts** Part 1 : Complexity analysis

Part 2

+ **0 pts** Not attempted

✓ **+ 2.5 pts** **Part 2 : Correct and detailed algorithm**

+ **1.25 pts** Part 2 : Partially correct algorithm or details not clear

✓ **+ 1.5 pts** **Part 2 : Correct proof**

+ **0.75 pts** Part 2: Partial or incomplete proof

+ **1 pts** Part 2 : Complexity analysis

Part 3

+ **0 pts** Not attempted

✓ **+ 2.5 pts** **Part 3 : Correct and detailed algorithm**

+ **1.25 pts** Part 3 : Partially correct algorithm or details not clear

✓ **+ 1.5 pts** **Part 3 : Correct proof**

+ **0.75 pts** Part 3 : Partial or incomplete proof

+ **1 pts** Part 3 : Complexity analysis

+ **0 pts** Not attempted

+ 2 Point adjustment

m and n in complexity analysis undefined

3. **Forex Trading**

(a) The given graph has edge weights R(i,j) between vertices i and j

We define a Graph G which can be obtained just by replacing the edges between i and j by a new value W(i,j) = -log(R(i,j)).

The product in the given graph,

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

Taking log on both sides,

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

Taking product with -1 ,

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

Replacing R(i,j) with W(i,j) ,

$$W[i_1, i_2] + W[i_2, i_3] + ... + W[i_{k-1}, i_k] + W[i_k, i_1] < 0$$

which implies a negative weight cycle in the graph G is equivalent to finding a cycle such that exchanging money over the same cycle results in positive gain in our given graph.

We can use the Bellman Ford Algorithm to detect a negative weight cycle.

To find a negative cycle we initially add a dummy source vertex (let it be s) and define a new graph G' such that we start with Graph G and add a new node s to it, and connect s to each other node v in the graph via an edge of cost 0 that W[s,v] = 0.

Claim 1: The graph G has a negative cycle C such that there is a path from the source s to C if and only if the original graph has a negative cycle.

Proof. Assume G has a negative cycle. Then this cycle C clearly has an edge from s in G, since all nodes have an edge from s. Now suppose G has a negative cycle with a path from s. Since no edge comes into s in G, this cycle cannot contain s. Since G is the same as G aside from the node s, it follows that this cycle is also a negative cycle of G. □

Claim 2: The graph G has a negative weight cycle if and only if we can make improvement in vector D even in n^{th} round where n is the number of vertices.

If G' has no negative cycles, then there is a shortest path from s to t that is simple (i.e., does not repeat nodes), and hence has at most n - 1 edges.

Part 1: Improvement in vector D even in n^{th} round where n is the number of vertices implies the graph G has a negative weight cycle.

Proof. Proof by contradiction :

Let us suppose it doesn't has any negative weight cycle.

Since every cycle has non-negative cost, the shortest path P from s to the vertex x which was improved in the n^{th} round with the fewest number of edges does not repeat any vertex v . For if P did repeat a vertex v, we could remove the portion of P between consecutive visits to v, resulting in a path of no greater cost and fewer edges. But since the number of vertices are n , it must repeat a vertex .

Hence contradiction. □

Part 2: the graph G has a negative weight cycle implies Improvement in vector D even in n^{th} round where n is the number of vertices.

Proof. Proof by contradiction :

Let us suppose there was no improvement in n^{th} round. But it implies that there will not be any improvement in any of the further rounds as the D vectors remain the same. But this contradicts the fact that there is a negative cycle in the graph as this implies D value of certain vertices must become arbitrarily negative.

Hence contradiction.

Hence proved □

Time Complexity: O(mn)

Proof. We do n iterations in which we go over all the edges ,where n is the number of vertices in G' and m is the number of edges in G'.

□

Algorithm 5: Forex Trading A

```
for v ∈ V do
    D[v] = inf and parent[v] = null ;
end
D[s] ← 0 ;
for i ← 1 to ||V|| − 1 do
    foreach (x, y) ∈ E do
        if D[y] > D[x] + W[x,y] then
            D[y] ← D[x] + W[x,y] ;
            parent[y] ← x ;
        end
    end
end
foreach (x,y) ∈ E do
    if D[y] > D[x] + W[x,y] then
        return "Negative Cycle Found";
    end
end
return D,parent;
```

(b) If there is a negative weight cycle , we can extract the cycle in the following way :

We know if there is a negative cycle, then there will be an improvement in the n_{th} round.

Let the vertex being relaxed be x (any of those which get relaxed).

Claim 1: x will either lie on a negative weight cycle or it will be reachable from it.

Proof. Sine it has been updated in the nth iteration it means that the updating path is at least length n which implies that there must be a cycle in that path as there cannot be a simple path of length n+1 in a graph consisting of n vertices. Also the cycle has to be a negative weight cycle otherwise we can remove the cycle and be better off.

Hence proved.

□

Claim2: If we move along the parents of vertices starting from x n times we will reach a node belonging to a cycle.

Proof. Suppose we haven't reached a vertex belonging to a cycle then we have traversed a simple path of length n+1 in a graph consisting of n vertices which is a contradiction.

Also when we reach a vertex in a cycle , the parents of the vertex will also belong to the cycle otherwise it is not a cycle.

Hence after n iterations we will reach a node belonging to a cycle.

□

From claims 1 and 2, we devise the algorithm as follows :

i. Let C be the negative cycle. To get a vertex in C, starting from the vertex x, we go through the parents of the vertex n times to reach a vertex say $v \in C$.
ii. Now, we go from this vertex, through it's parents, adding the vertices in a list until we reach the same vertex v.
iii. We then return the reverse of the list which contains a negative weight cycle.

Time Complexity: O(mn)

Proof. Bellman-Ford is O(mn). We do one more iteration over the edges to determine the vertex v which will be changed in the nth iteration. Since a cycle can have at most n vertices, retrieving the cycle via the parent pointers is also O(n). The total time complexity is O(mn) where m is the number of edges and n is the number of vertices.

□

Algorithm 6: Forex Trading B

```
1  for v ∈ V do
2  |  D[v] = inf and parent[v] = null ;
   end
3  D[s] ← 0 ;
4  for i ← 1 to ‖V‖ − 1 do
5  |  foreach (x, y) ∈ E do
6  |  |  if D[y] > D[x] + W[x,y] then
7  |  |  |  change ← y ;
8  |  |  |  D[y] ← D[x] + W[x,y] ;
9  |  |  |  parent[y] ← x ;
   |  |  end
   |  end
   end
10 foreach (x,y) ∈ E do
11 |  change ← -1 ;
12 |  if D[y] > D[x] + W[x,y] then
13 |  |  change ← y ;
   |  end
   end
14 if change = -1 then
15 |  print("No negative cycle");
   end
16 for i ← 1 to ‖V‖ − 1 do
   |  change ← parent[change] ;
   end
17 y ← parent[change] ;
18 Cycle ← { change } ;
19 while y != change do
20 |  Cycle.append(y) ;
21 |  y ← parent[y]
   end
22 return reverse(C);
```

3 Forex Trading 14.5 / 15

✓ - 1 pts Define the graph, vertices and edge details, and also the number and weights of the edges.(2)

+ 0.5 Point adjustment

4. **Coin Change**

(a) Recursion :

Let ways (i, j) be the number of ways to make change for Rs. i, given an infinite amount of coins/notes of denominations, D[1], D[2] ,. . . , D[j].

The possibilities are:

i. The amount Rs. i is formed by taking at least one coin of denomination D[j]

ii. The amount Rs. i is formed by taking coins from denomination D[1] , D[2] , ... , D[j-1].

The above cases are trivially exhaustive.

The base cases are:

i. We can only make Rs. 0 using 0 denomination and in just 1 way. So , ways(0, 0) = 1

ii. We cannot make Rs. x more than 0 using 0 denomination. So, ways(x, 0) = 0 $\forall x,\ 0 < x \leq n$

The final recursion is as follows :

$$ways(i,j) = \begin{cases} 1, & if \text{ i} = 0 \text{ and j} = 0 \text{ .} \\ 0, & if \text{ i} > 0 \text{ and j} = 0 \text{ .} \\ ways(i, j-1), & if \text{ i} < \text{D[j] .} \\ ways(i - D[j], j) + ways(i, j-1) & otherwise \end{cases}$$

Proof of Correctness of Algorithm:

Let a hypothesis H(i,j) is that ways(i,j) is true.

Proof by strong induction :

The Base cases are correct as explained above.

Induction Hypothesis : Let H(a,b) is true ∀ (a,b) such that (a <= i, b <= j and (a,b) != (i,j))

Induction step :

Since we use the recursion relation defined above and the recursion relation is correct and the value of ways(i,j) only depends on ways(a,b) such that (a <= i, b <= j and (a,b) != (i,j)).

Hence H(i,j) is true from induction hypothesis.

Hence proved

Algorithm 7: Coin Change A

```
1 ways ← array of size (n+1)*(k+1) all of the elements initialised to 0;
2 D ← { d1 , d2 , d3 , ... , dk } ;
3 ways[0][0] ← 1 ;
4 for i ← 0 to n do
5 |   for j ← 1 to k do
6 |   |   if i < D[j] then
7 |   |   |   ways[i][j] ← ways[i][j-1];
  |   |   else
8 |   |   |   ways[i][j] ← ways[i][j-1] + ways[i-D[j]][j];
  |   |   end
  |   end
  end
9 return ways[n][k];
```

Time Complexity : O(n*k)

Line 1 : To initialise the array we take O(n*k) time. Line 2 : Declare an array D consisting of all the denominations Line 4 - 8 : We run two loops one for the amount of money i which loops for 0 to n and the other for the restriction of the number of denominations used which will again take O(n*k) as each iteration in the inner loop takes O(1) time.

Space Complexity : O(n*k)

We have to declare a 2D array ways which has size (n+1)*(k+1)

(b) Recursion :

Let changes (i,j) be the minimum number of coins to find a change of Rs. i given an infinite amount of coins/notes of denominations, D[1], D[2] ,. . . , D[j].

The possibilities are:

i. The amount Rs. i is formed by taking at least one coin of denomination D[j]

ii. The amount Rs. i is formed by taking coins from denomination D[1] , D[2] , ... , D[j-1].

The above cases are trivially exhaustive.

The base cases are:

i. We can only make Rs. 0 using 0 denomination and using 0 coins. So , changes(0,0) = 0

ii. We cannot make Rs. x more than 0 using 0 denomination. So , changes(x,0) = inf (Note : inf means infinity) for all x > 0 and x <= n

The final recursion is as follows :

$$changes(i,j) = \begin{cases} 0, & if \text{ i} = 0 \text{ and j} = 0 . \\ inf, & if \text{ i} > 0 \text{ and j} = 0 . \\ changes(i, j-1), & if \text{ i} < \text{D[j]} . \\ min(changes(i - D[j], j) + 1, changes(i, j-1)) & otherwise \end{cases}$$

Proof of Correctness of Algorithm:

Let a hypothesis H(i,j) is that changes(i,j) is true.

Proof by strong induction :

The Base cases are correct as explained above.

Induction Hypothesis : Let H(a,b) is true ∀ (a,b) such that (a <= i, b <= j and (a,b) != (i,j))

Induction step :

Since we use the recursion relation defined above and the recursion relation is correct and the value of changes(i,j) only depends on changes(a,b) such that (a <= i, b <= j and (a,b) != (i,j)).

Hence H(i,j) is true from induction hypothesis.

Hence proved

Time Complexity : O(n*k)

Line 1 : To initialise the array we take O(n*k) time. Line 2 : Declare an array D consisting of all the denominations Line 4 - 8 : We run two loops one for the amount of money i which loops for 0 to n and the other for the restriction of the number of denominations used which will again take O(n*k) as each iteration in the inner loop takes O(1) time. Line 13 - 17: We retrieve the answer by walking backwards. Since in every iteration either balance or D_left decreases by at least 1 which implies the time complexity of this step is O(n+k)

Space Complexity : O(n*k)

We have to declare a 2D array changes which has size (n+1)*(k+1)

Algorithm 8: Coin Change B

```
changes ← array of size (n+1)*(k+1) all of the elements initialised to inf;
D ← { d1 , d2 , d3 , ... , dk } ;
changes[0][0] ← 0 ;
for i ← 0 to n do
    for j ← 1 to k do
        if i < D[j] then
            changes[i][j] ← ways[i][j-1];
        else
            changes[i][j] ← min( changes( i-D[j] , j) + 1 , changes(i,j-1) );
        end
    end
end
solution = { } ;
if changes[n][k] = inf then
    Print ( " No change exists " ) ;
end
balance ← n ;
D_left ← k ;
while balance > 0 and D_left > 0 do
    if changes[balance][D_left] = changes[balance][D_left-1] then
        D_left ← D_left - 1;
    else
        solution.append(D[D_left]);
        balance ← balance - D[D_left];
    end
end
return solution;
```

4 Coin Change 11 / 15

Part a

✓ **+ 3 pts** **Correct and detailed algorithm**

+ 2 pts Correct but not a detailed algorithm/Minor mistakes in the algorithm

✓ **+ 1 pts** **Complexity analysis**

✓ **+ 3 pts** **Correct Proof of Correctness of the algorithm**

+ 2 pts Minor mistakes in the Proof of Correctness of the algorithm/Not a detailed proof

+ 0 pts Incorrect algorithm

Part b

+ 3 pts Correct and detailed algorithm

✓ **+ 2 pts** **Correct but not a detailed algorithm/Minor mistakes in the algorithm**

+ 1 pts Complexity analysis

+ 4 pts Correct Proof of Correctness

✓ **+ 3 pts** **Minor mistakes in the Proof of Correctness of the algorithm/Not a detailed proof**

+ 0 pts Incorrect algorithm

- 1 Point adjustment