

COL351 Assignment2

Aayush Goyal
2019CS10452

Aniket Gupta
2019CS10327

September 2021

1 Algorithms Design book

We will denote $\sum_{i \in S_1} x_i$, $\sum_{i \in S_2} x_i$ and $\sum_{i \in S_3} x_i$ by sum_1 , sum_2 and sum_3 respectively. Also, the sum of no. of questions in first i chapters is denoted by $\text{totalQuestions}(i)$.

Now, the basic idea is to check if it is possible to partition the chapters in three parts for all the possible values of sum_1 , sum_2 and sum_3 and then chose the best among them.

We can write the recursive relation to check if such a partition exists for a given value of n , sum_1 , sum_2 and sum_3 as:

$$\begin{aligned} \text{partition}(n, \text{sum}_1, \text{sum}_2, \text{sum}_3) &= \text{True} && \text{if } (n = \text{sum}_1 = \text{sum}_2 = \text{sum}_3 = 0) \\ &= \text{False} && \text{if } ((n = 0) \text{ and } (\text{sum}_1 \neq 0 \text{ or } \text{sum}_2 \neq 0 \text{ or } \text{sum}_3 \neq 0)) \\ &&& \text{or } \text{sum}_1 < 0 \text{ or } \text{sum}_2 < 0 \text{ or } \text{sum}_3 < 0 \\ &&& \text{or } \text{sum}_1 + \text{sum}_2 + \text{sum}_3 \neq \text{totalQuestions}(n) \\ &= \text{partition}(n-1, \text{sum}_1 - x_n, \text{sum}_2, \text{sum}_3) \\ &\quad \text{or } \text{partition}(n-1, \text{sum}_1, \text{sum}_2 - x_n, \text{sum}_3) \\ &\quad \text{or } \text{partition}(n-1, \text{sum}_1, \text{sum}_2, \text{sum}_3 - x_n) \end{aligned}$$

In the above recursive relation, $\text{partition}(n, \text{sum}_1, \text{sum}_2, \text{sum}_3)$ returns True if there exists a partition for the first n chapters such that the sums of the no. of questions in the three sets are sum_1 , sum_2 and sum_3 respectively and False otherwise. Also, note that the or and and in the above equation are logical operators. Explanation of the above recursive relation:

The first line of the equation considers the case when no. of chapters is zero. In that case, the only possible partition is when sum of questions in each set is zero.

The second line consider the basic cases when there is no such partition.

The third line considers the cases when the n^{th} chapter is considered in one of the three sets. For example, if the n^{th} chapter is taken in set S_1 , then $\text{partition}(n, \text{sum}_1, \text{sum}_2, \text{sum}_3)$ is equivalent to saying that if it is possible to partition the first $n-1$ chapters with sums $\text{sum}_1 - x_n$, sum_2 and sum_3 respectively, i.e. $\text{partition}(n-1, \text{sum}_1 - x_n, \text{sum}_2, \text{sum}_3)$. Similarly, consider the case if n^{th} chapter is considered in S_2 and S_3 . Taking the logical or ensures that if any of these three cases is True, then the original query is also True.

Observation:

Note that in the above equation, for the cases when partition is possible, values of sum_1 and sum_2 uniquely determines the value of sum_3 . The value of sum_3 in that case is $\text{totalQuestions}(n) - \text{sum}_1 - \text{sum}_2$. Thus, the above equation can be equivalently rewritten as:

$$\begin{aligned} \text{partition}(n, \text{sum}_1, \text{sum}_2) &= \text{True} && \text{if } (n = \text{sum}_1 = \text{sum}_2 = 0) \\ &= \text{False} && \text{if } ((n = 0) \text{ and } (\text{sum}_1 \neq 0 \text{ or } \text{sum}_2 \neq 0)) \\ &&& \text{or } \text{sum}_1 < 0 \text{ or } \text{sum}_2 < 0 \\ &&& \text{or } \text{sum}_1 + \text{sum}_2 > \text{totalQuestions}(n) \\ &= \text{partition}(n-1, \text{sum}_1 - x_n, \text{sum}_2) \\ &\quad \text{or } \text{partition}(n-1, \text{sum}_1, \text{sum}_2 - x_n) \\ &\quad \text{or } \text{partition}(n-1, \text{sum}_1, \text{sum}_2) \end{aligned}$$

The above equation is equivalent to the previous equation, but there is one less variable in the new equation. Once, all the valid values of sum_1 and sum_2 , that gives a possible partition, are determined, we check for the partition with minimum value of $maxsum_1, sum_2, sum_3$ (Note: $sum_3 = totalQuestions(n) - sum_1 - sum_2$) and return that partition. Also, note that for each valid value of sum_1 and sum_2 , we also need to store the set in which the current chapter is included to return the final partition.

This recursive relation can be implemented using dynamic programming as shown below:

Algorithm:

2 Course Planner

Problem Formulation:

This problem can be reformulated in a directed graph problem. Construct a directed graph with vertex set C and edge set E such that $(x,y) \in E$ (directed edge from x to y) iff x is a prerequisite of y , i.e. $x \in P(y)$.

Claim 1:

If there is any path $c_1 -> c_2 -> c_3 \dots -> c_{t-1} -> c_t$ in the above defined directed graph, then the courses must be done in the order $c_1, c_2, c_3 \dots, c_{t-1}, c_t$.

Proof for claim 1:

According to the definition of the edge in the above graph, c_i is a prerequisite of c_{i+1} in this path $\forall i \in (1, 2, 3, \dots, t-1)$ and thus course c_{i+1} can be taken only if c_i is already completed. Thus, the way in which the edges are defined imply that the courses must be done in the order $c_1, c_2, c_3 \dots, c_{t-1}, c_t$.

2.1

Claim 2: An order for taking the courses, so that a student is able to take all the n courses with the prerequisite criteria satisfied, exists if and only if there is no cycle in the above defined directed graph.

Proof:

Proof for forward implication:

We prove "If there is required ordering available, then there is no cycle in the graph defined" by using contradiction.

Let us assume such an ordering exists but there is a cycle in the directed graph. Let that cycle be $c_1 -> c_2 -> c_3 \dots c_n -> c_1$. Then by claim 1, the order of completion must follow $c_1, c_2, c_3 \dots c_k, c_1$. This implies that c_1 must be completed before c_1 which is not possible. Therefore, there is no such ordering and it contradicts our assumption. Hence, there cannot be a cycle in the directed graph defined if there is a required ordering available.

Proof for backward implication:

We prove "If there is no cycle in the graph defined, then there is a required ordering available".

Run a dfs traversal on the directed graph defined. Since the graph is acyclic, therefore there will be no back-edges in the dfs traversal tree of the graph. Now, arranging the courses in decreasing order of finish time, we get an ordering (topological ordering) such that for all edges (u,v) in the graph, u occurs before v in the ordering. Now, we will prove inductively that taking the courses in the topological ordering obtained, a student can complete all the courses with prerequisites completed.

Let the topological ordering be $v_1, v_2, v_3, \dots, v_n$ Induction predicate:

$H(i) \Rightarrow$ If we take the courses in topological ordering, then at the time when we take course v_i , all its prerequisites will be already satisfied. Base Case:

Since, in a topological ordering, all the edges go from left to right therefore there is no node which has an from itself to v_1 . Thus, there is no prerequisite for course v_1 and it can be taken in the beginning.

Induction Step:

Let us assume that $H(j)$ is true for all $0 < j \leq i$ for some i ($0 < i \leq n$). Then, we will prove that $H(i+1)$ is also

true.

Since in a topological ordering, all the edges are from left to right therefore, all the nodes which an edge from itself to v_{i+1} will lie before v_{i+1} in the topological ordering. From our assumption, all the prerequisites of v_{i+1} will be already satisfied when it is visited in the topological order and thus $H(i+1)$ is also true.

Thus, we proved that "If there is no cycle in the graph defined, then there is a required ordering available". This completes the proof for claim 2.

Algorithm:

The basic idea is to check if the graph obtained is a directed acyclic graph and if it is so, find a topological ordering for the graph.

Algorithm:

Time Complexity:

Creating the adjacency list (outAdj) and edge set from the given problem will take $O(m)$ time where m is the no of (prerequisite, course) pairs in the original problem. If the no. of courses are n , then dfs traversal will take $O(n+m)$ time. Checking for back edges takes $O(m)$ time. Then, sorting the nodes according to decreasing order of endTime can be done using bucket sort in $O(n)$ time. Thus, the overall time complexity of the algorithm is $O(n+m)$.

2.2

We are assuming that a valid ordering exists which allows to complete all the courses with the prerequisites complete. If there is no such ordering, it can be detected using algorithm in part 2.1

Claim 3: For any course c , the minimum no of semesters required to complete that course is one more than the length of longest path ending at c in the directed graph obtained.

Proof for claim 3:

Since we are assuming that an ordering exist, then there is graph is acyclic and a topological ordering exist (by claim 2). Let the topological ordering obtained be $v_1, v_2, v_3, \dots, v_n$. We prove claim 3 by induction on the topological ordering.

Induction Predicate: $H(i) :=$ minimum no. of semesters required to complete course v_i is one more than length of longest path ending at v_i .

Base Case: Since in a topological ordering, all the edges go from left to right, the longest path ending at v_1 has length 0. Now, since there is no. node which has an edge from itself to v_1 , v_1 has no pre-requisite and thus it can be completed in the first semester itself which is one greater than length of longest path ending at v_1 . Thus, $H(1)$ is true.

Induction Step:

Let us assume that $H(j)$ is true for all $0 < j \leq i$ for some i ($0 < i \leq n$). We will prove that $H(i+1)$ is also true. Now the minimum no. of semesters required to complete the course v_{i+1} is obtained if v_{i+1} is taken in the very next semester when all its prerequisites are completed. Let the prerequisites of v_{i+1} be $v_{k_1}, v_{k_2}, v_{k_3}, \dots, v_{k_t}$. According to the way the graph edges are defined, there is an edge from each of these prerequisites to v_{i+1} . Thus, in the topological ordering, all these nodes appear before v_{i+1} . Among all the prerequisites of v_{i+1} , say v_{k_l} has the longest path ending at that node, say of length l' . Then, v_{k_l} will take $l'+1$ semesters for its completion according to induction hypothesis. Also, minimum no. of semesters for the other prerequisites will be less than or equal to $l'+1$ due to the induction hypothesis and the assumption that among the prerequisites of v_{i+1} , v_{k_l} has the longest path ending at that node. Now, the minimum no. of semesters to complete v_{i+1} will be $l'+2$. Also, all the paths that end at v_{i+1} will have one of the prerequisites as the second last node in the path (if path length is greater than 1). Thus the length of longest path ending at v_{i+1} is one more than the length of longest path ending at any of its neighbours. Therefore, length of longest path ending at v_{i+1} is $l'+1$. Thus minimum no. of semesters required to complete v_{i+1} is one more than the length of longest path ending at v_{i+1} [$l'+2 = (l'+1)+1$]. Thus, $H(i+1)$ is also true.

Thus, by principle of induction, we proved the given claim.

Now, the minimum no. of semesters required to complete all the courses is one more than length of the longest path ending at any of the course. Thus, we need the length of longest path in the directed graph obtained.

Algorithm:

Correctness of algorithm:

For this part, I am assuming that the graph has no cycle. If there is a cycle in the graph, then it can be detected using algorithm in part 2.1. First we obtain a topological ordering of the courses such that for any edge $(u \rightarrow v)$, u occurs before v in the ordering. We prove that this algorithm gives longest path ending at all the courses in the graph obtained.

To prove this, we show that the loop invariant (for loop in line 11-14)- "After i^{th} iteration of outer loop, length of longest path ending at first i courses in the topological ordering is correctly calculated in longestPath[] array".

We prove this by induction on the above loop invariant mentioned.

Let the topological ordering obtained be $v_1, v_2, v_3, \dots, v_n$. Base Case:

Before the first iteration, $\text{longestPath}[v_1] = 0$. Also, since for any edge $(u \rightarrow v)$, u occurs before v in the ordering, there is no node which has an edge from itself to v_1 . Thus, the inner loop (line 12-14) is executed for zero times and therefore after the first iteration, $\text{longestPath}[v_1]$ is still zero, which is precisely the length of longest path ending at v_1 .

Induction step:

Let us assume that $\text{longestPath}[j]$ is correctly computed for all $0 < j \leq i$ for some i ($0 \leq i < n$) before the $(i+1)^{th}$ iteration of the outer loop. Then, during the $(i+1)^{th}$ iteration, all the prerequisite courses are visited in the inner loop. Since the nodes are being traversed in topological order, all its prerequisite courses of v_{i+1} will appear before v_{i+1} in the ordering and thus the longest path for these prerequisite course nodes is correctly calculated due to the induction assumption. Now, length of longest path ending at v_{i+1} is one more than length of longest path ending at any of its prerequisite. Thus, the line 12-14 correctly computes the length of longest path ending at v_{i+1} . Thus, after the $(i+1)^{th}$ iteration of the outer loop, length of longest path ending at v_{i+1} is correctly calculated.

Thus, by principle of induction, the loop invariant holds after each iteration and thus at the end of longest path ending at all the courses is correctly obtained.

Now, the line 16-18 finds the length of longest path ending at any node in the graph. Then line 20 increases the sol by 1 since minimum no. of semesters required is one more than the length of the longest path in the graph.

Thus, the given algorithm is correct.

Time complexity:

Finding the topological ordering takes $O(n+m)$ time. Now, the inner for loop in lines 12-14 visits prerequisite courses of the current course taken in the outer for loop. Thus, the total time taken for line 11-14 is $O(n+m)$ in the graph (n is total no. of courses and m is the total no. of edges in the graph obtained). Also, the loop in line 17-18 takes $O(n)$ time. Thus, total time complexity is $O(n+m)$.

2.3

Claim 4: For two courses c and c' , $L(c) \cap L(c')$ is empty if and only if there is no node from which there is a path to both c and c' in the directed graph obtained.

Proof for claim 4:

If there is a path from any node v' to another node v'' , then v' must be completed before v'' (see claim 1). Thus, $v' \in L(v'')$.

If $L(c) \cap L(c')$ is empty, then there is no node, say v , from which there is a path to both c and c' .

We prove this by contradiction. Let us assume that $L(c) \cap L(c')$ is empty but there exist a node v from which there is path to both c and c' . Then, according to claim 1, v must be completed before c and v must be

completed before c' as well. Thus, by definition of $L(c)$, $v \in L(c)$ and $v \in L(c')$ which implies that $v \in L(c) \cap L(c')$ which contradicts the fact that $L(c) \cap L(c')$ is empty. Thus there is no such node v from which there is a path to both c and c' .

If there is no node from which there is a path to both c and c' then $L(c) \cap L(c')$ is empty.

We prove this by contradiction. Let us assume that

3 Forex Trading

The problem can be visualized as a directed graph, where if there is an edge from node ' i ' to node ' j ' then, its cost is represented by $R(i,j)$. For the problems below we will be converting the weight of edges into another form. Every edge weight will be changed to $R^*(i,j)$, where $R^*(i,j) = -\log(R(i,j))$

3.1

We need to find a cycle whose edges satisfy the property that the product of reward of all its edges is greater than 1. Thus if the cycle contains the vertices $i_1, i_2, \dots, i_k, i_{k+1} = i_1$, then we need a cycle such that the rewards satisfy $R[i_1, i_2] * R[i_2, i_3] * \dots * R[i_k, i_1] > 1$. In this relation, take $-\log$ on both sides. Since \log is an increasing function, we have the following $-\log(R[i_1, i_2] * R[i_2, i_3] * \dots * R[i_k, i_1]) < -\log(1)$. This can be written as $(-\log(R[i_1, i_2])) + (-\log(R[i_2, i_3])) + \dots + (-\log(R[i_k, i_1])) < 0$. Each of the terms on the lhs can be written as $R^*[i,j]$ (the changed reward we defined above). Now the problem becomes to find if there is any cycle with total weight negative in the graph with new weighted edges, given by $R^*[i,j] = -\log(R[i,j])$. This becomes a standard problem now to determine if there is any negative weight cycle in the graph or not. This can be done by modifying the Bellman-Ford algorithm a little bit.

Claim 1: G contains a 'negative weight cycle' if and only if we are able to make an improvement even in the Distance vector even in the n^{th} round.

Proof of Claim 1:

Algorithm Outline:

We will use the Bellman-Ford algorithm to determine if there is any negative weight cycle in the graph or not. The basic idea is, after running the $n-1$ ($n = |V|$) iterations of the Bellman-Ford algorithm, if the graph contains a negative weight cycle then by Claim 1, we are able to make an improvement even in the n^{th} iteration. The graph here is assumed to be connected since it represents the exchange rates between currencies.

Algorithm:

```

1 edges <- contains the edge list
2 R <- the weight of edges, R[i,j] denotes the weight of edge from i to j
3 R* <- this will store the modified edge weights
4
5 distance <- this contains the distance of each node from the starting vertex
6               (Every distance is initialized by infinity)
7
8 N <- |V|      # number of edges in the graph
9
10
11 # Making the R* vector
12 for i <- 0 to N-1 do:
13     for j <- 0 to N-1 do:
14         R*[i,j] = -1*log(R[i,j])
15
16
```

```

17 # initialing the distance vector
18 for i<-0 to N-1 do:
19     distance[i] <- inf
20
21 start <- start is the starting node
22
23 # Now doing the first N-1 iterations of bellman ford algorithm
24 for i<- 1 to n-1 do:
25     for edge in edges do:
26         u = edge.first # the starting vertex
27         v = edge.second # the ending vertex, means edge from u to v
28         w = R*[u,v]
29         if (distance[v] > distance[u] + w) then:
30             distance[v] = distance[u] + w
31
32 # Now doing the Nth iteration to find if we are still able to make any
33 # improvement, if we are able to make an improvement that means the graph
34 # has a negative weight cycle.
35
36 hasNegCycle <- False
37
38 for edge in edges do:
39     u = edge.first # the starting vertex
40     v = edge.second # the ending vertex, means edge from u to v
41     w = R*[u,v]
42     if (distance[v] > distance[u] + w) then:
43         hasNegCycle <- True
44         break
45
46 # if hasNegCycle is True that means we found a negative weight cycle in the
47 # graph. This means there exists a cycle which has product of the R[i,j] of
48 # edges greater than 1.

```

Time complexity analysis:

The time complexity of the following algorithm is $O(mn)$, n is the number of vertices, m is the number of edges which here is $O(n^2)$. The breakout of complexity is as follows:

1. Making R^* from takes $O(n^2)$ time since we iterate with 2 nested for loops.
2. Now the main part of algorithm from line 24-30, we run the outer loop $n - 1$ times and in each for loop, we iterate over the entire edge set and thus in the inner loop we do m iterations. Thus overall we do $(n - 1) * m$ iterations.
3. In line 38-44 we do m iterations, since we just iterate over the entire list once
4. Now combining the point 2 and 3 above we have $n * m$ iterations, and thus it takes $O(nm)$ time.

Thus the overall time complexity of the process becomes $O(mn + n^2)$ and if we consider this as a complete graph then $m = O(n^2)$ and hence the overall time complexity is $O(n^3)$

3.2

In the above we used Bellman ford algorithm to determine if there exists a cycle with negative edge weight. In this part we will print the cycle if it exists and -1 if it doesn't exist.

Algorithm outline:

We will need to store the parent of each vertex while we are running the iterations of bellman ford algorithm (ie during the first $n-1$ iterations). Now when we are running the n^{th} iteration we will need to store the first vertex where we encountered an improvement. That vertex is part of the negative cycle. Now from that vertex we can keep storing the parents of it untill we comeback to the vertex where we started.

Claim 1: The first vertex where we detect improvement is part of the negative weight cycle

Proof of Claim 1:

Claim 2: Backtracking till we reach the vertex found using claim 1 again will give us the cycle.

Proof of Claim 2:

Algorithm:

```

1  edges <- contains the edge list
2  R <- the weight of edges, R[i,j] denotes the weight of edge from i to j
3  R* <- this will store the modified edge weights
4
5  distance <- this contains the distance of each node from the starting vertex
6                (Every distance is initialized by infinity)
7
8  parent <- this is to store the parent of each vertex
9
10 N <- |V|    # number of edges in the graph
11
12
13 # Making the R* vector
14 for i<-0 to N-1 do:
15     for j<-0 to N-1 do:
16         R*[i,j] = -1*log(R[i,j])
17
18
19 # initialing the distance and parent vector
20 for i<-0 to N-1 do:
21     distance[i] <- inf
22     parent[i] <- -1
23
24
25 start <- start is the starting node
26
27 # Now doing the first N-1 iterations of bellman ford algorithm
28 for i<- 1 to n-1 do:
29     for edge in edges do:
30         u = edge.first # the starting vertex
31         v = edge.second # the ending vertex, means edge from u to v
32         w = R*[u,v]
33         if (distance[v] > distance[u] + w) then:
34             distance[v] = distance[u] + w
35             parent[v] = u
36
37

```

```

38 # Now doing the Nth iteration to find if we are still able to make any
39 # improvement, if we are able to make an improvement that means the graph
40 # has a negative weight cycle.
41
42 hasNegCycle <- False
43 cycleVertex <- -1
44
45 for edge in edges do:
46     u = edge.first # the starting vertex
47     v = edge.second # the ending vertex, means edge from u to v
48     w = R*[u,v]
49     if (distance[v] > distance[u] + w) then:
50         hasNegCycle <- True
51         cycleVertex <- v
52         break
53
54 # if hasNegCycle is True that means we found a negative weight cycle in the
55 # graph. This means there exists a cycle which has product of the R[i,j] of
56 # edges greater than 1 and cycleVertex is one of the vertices of the cycle.
57
58 # if cycleVertex is -1 then we don't have any negative cycle and we end the
59 # program here
60
61 if (cycleVertex == -1) then:
62     End the procedure
63
64 # Now backtracking using the parent array to detect cycle
65 cycle <- [] # empty list that store the vertices of the cycle
66
67 cycle.append(cycleVertex)
68 v = parent[cycleVertex]
69
70 while True do:
71     cycle.append(v)
72     if (v == cycleVertex) then:
73         break
74     else:
75         v = parent[v]
76
77 cycle = reverse(cycle) # reversing the vertices present in the cycle
78
79 # We have vertices stored in the cycle vector and if the elements of cycle are
80 # c1, c2, ..., ck, ck+1 = c1 then edges that are included in the graph are
81 # c1->c2, c2->c3, ..., ck->c1

```

Time Complexity Analysis:

The overall time complexity of the following algorithm will be $O(n^3)$. Most of the part is same as that of 3a, where we are running the Bellman ford Algorithm.

1. We are storing the parent during each iteration and this takes $O(mn)$ time.
2. When running the n^{th} iteration we need to find the vertex which is a part of the negative cycle. This is done in $O(m)$ time.

3. Next we run the for loop in the lines 70 to 75. This is a while True loop but it will end as soon as we have encountered the first vertex again and hence this can atmost n times. Thus the time complexity of this part is also $O(n)$.

Hence the overall time compexity will be $O(mn)$ and since m is $O(n^2)$, the time complexity becomes $O(n^3)$

4 Coin Change

4.1

We have coins of k denomations, and the vector d which stores the values of that denomination. $d[i]$ means the value of the i^{th} currency. Also we have been given an infinite supply of the coins. We need to find the number of ways in which we can use the currency to sum up to n .

The problem can be solved using dynamic programming. We will assume that since we need to make the change, it doesn't matter in what order do we pick the coins.

4.1.1 Algorithm outline

To make sure that we are not making any repetitions, that is the coins are being considered in unordered manner, we will count the number of ways to make some amount denomination by denomination. First we will see the number of ways to make different amounts from 0 to n using $d[0]$ and store that in a *ways* vector. Next we will update the array using $d[1]$ and so on till $d[k]$.

4.1.2 Algorithm

```

1 d[k] <- this contains the denomination values
2
3 ways[n+1] <- this contains number of ways we can make a particular amount
4
5 for i <- 0 to n do:
6     ways[i] = 0      # initially no ways to make any amount
7 ways[0] =1          # there is only one way, ie take no coin
8
9 for i <- 0 to k-1 do:
10     for j <- 1 to n do:
11         if (j-d[i] >=0) then:
12             ways[j] += ways[j-d[i]]
13
14 return ways[n]
15 # ways[n] contains the number of ways we can make chang amount n

```

4.1.3 Proof of correctness

4.1.4 Time complexity Analyssis

The time complexity of the above algorithm will be $O(nk)$ since we running 2 nested for loops, the outer loop runs for n iterations and the inner loop runs for k iterations. Thus overall the program runs for nm iterations. Thus the time complexity is $O(nk)$

4.1.5 Space Complexity Analysis

The space required is for storing count of coins required for a certain denomination. It needs $O(n)$ space. The other space is required to store the denominations, which need $O(k)$ space. Thus the overall space complexity is $O(n + k)$.

4.2

The problem can be thought in a recursive manner. A certain amount has to be made from some other amount by using more coin. So we can take a minimum over all the possibilities and then consider the possibility in which least number of coins are needed.

4.2.1 Algorithm outline

We will maintain an array count in which we will store the number of coins needed to make a certain amount. Next we know that no coin is needed to make the amount 0 and hence $\text{count}[0] = 0$. Then we iterate for every amount to find the minimum number of coins required to make it. It is $1 + \text{count}[\text{amount} - \text{denomination value}]$ and we will take a minimum over all the possible denominations. This way in the end we will have the number of coins required to make amount n .

4.2.2 Algorithm

```
1 d[k] <- this contains the denomination values
2
3 count[n+1] <- this stores number of coins needed to make certain denomination
4
5 # initialize every value with -1
6 for i <- 0 to n do:
7     count[i] = INF # initially denoting it's not possible to form certain amt
8 count[0] = 0      # No coins are needed to form amount 0
9
10 for i <- 1 to n do:
11     for j <- 0 to k-1 do:
12         if (i - d[j] >= 0) then:
13             if (count[i-d[j]] < INF) then:
14                 count[i] = min(count[i], 1+ count[i-d[j]])
15
16 return count[n]
17 # this denotes minimum no. of coins required to form a certain amount
```

4.2.3 Proof of correctness

4.2.4 Time complexity Analysis

The time complexity of the above algorithm will be $O(nk)$ since we are running 2 nested for loops, the outer loop runs for n iterations and the inner loop runs for k iterations. Thus overall the program runs for nk iterations. Thus the time complexity is $O(nk)$.

4.2.5 Space Complexity Analysis

The space required is for storing count of coins required for a certain denomination. It needs $O(n)$ space. The other space is required to store the denominations, which need $O(k)$ space. Thus the overall space complexity is $O(n + k)$.