[CS202]-Algorithms I **2022-23**

Homework: 02

Name: Karan Sunil Kumbhar RollNo: 12140860 email: karansunilk@iitbhilai.ac.in

Collaborators Names:

Solution of problem 1.

Alogoritm

```
diameter = array of size n+1, initialized to all 0s

function bfs(init, arr, n):
q = empty queue
visited = array of size n+1, initialized to all 0s
for i in range(n+1):
    diameter[i] = 0

q.enqueue(init)
visited[init] = 1

while q is not empty:
    u = q.dequeue()

    for i in range(len(arr[u])):
        if visited[arr[u][i]] == 0:
            visited[arr[u][i]] = 1
            diameter[arr[u][i]] += diameter[u] + 1
            q.enqueue(arr[u][i])

max_node = 0
for i in range(n+1):
    if diameter[i] > diameter[max_node]:
        max_node = i

return max_node
function findDiameter(arr, n):
init = bfs(1, arr, n)
val = bfs(init, arr, n)
return diameter[val]

Driver code
n = number of nodes in the tree
arr = adjacency list representation of the tree

d = findDiameter(arr, n)
print("The diameter of the n-ary tree is", d)
```

Time complexity :- O(V+E)

Solution of problem 2.

To modify the Depth First Search (DFS) algorithm to check if a graph is bipartite, we can use the following approach:

Choose an arbitrary node as the starting node and assign it to one of the two sets, say set A. For each of its adjacent nodes, assign it to the other set, say set B. Recursively repeat step 2 for each adjacent node of the nodes in set B, assigning them to set A. If at any point we encounter a node that has already been assigned to the same set as one of its adjacent nodes, then the graph is not bipartite. If all nodes have been assigned to sets without any conflicts, then the graph is bipartite. This approach involves recursively traversing the graph using the DFS algorithm, but instead of simply marking the nodes as visited, we assign them to one of the two sets. We need to keep track of the set assignment of each node and ensure that we do not assign a node to the same set as any of its adjacent nodes.

Algorithm

```
DFS(G):
    for each vertex u     G.V:
    u.color = WHITE
    u.parent = NIL
    time = 0
    for each vertex u     G.V:
    if u.color == WHITE:
    DFS-VISIT(G, u)

DFS-VISIT(G, u):
    time = time + 1
    u.d = time
    u.color = GRAY
    for each v in G.Adj[u]:
        if v.color == WHITE:
        v.parent = u
        DFS-VISIT(G, v)
        else if v.color == GRAY:
        # Graph is not bipartite
        # Handle the situation accordingly
    u.color = BLACK
    time = time + 1
    u.f = time
```

Solution of problem 3.

We can prove that DFS terminates at the node from which every other node can be reached by contradiction.

Suppose DFS does not terminate at the node from which every other node can be reached. This means that there exists some node in the graph that has not been visited by DFS. But this contradicts the fact that there is at least one node from which every other node can be reached. If there is such a node, then DFS starting at that node should visit every other node in the graph.

Therefore, DFS must terminate at the node from which every other node can be reached in a directed graph. This is

because DFS explores all the vertices reachable from a given source vertex before moving to another vertex, and the vertex from which every other vertex is reachable will be reached by DFS during the exploration process.

Solution of problem 4.

To find the strongly connected components of a directed graph using DFS, we can use the Kosaraju's algorithm, which consists of the following steps:

Run DFS on the given graph and keep track of the finish times of each vertex. Transpose the graph, i.e., reverse the directions of all edges. Run DFS on the transposed graph in decreasing order of the finish times obtained in step 1. Each DFS tree in step 3 represents a strongly connected component of the original graph.

Pseudocode:

```
function Kosaraju(G):
    n = number of vertices in G
    visited = array of size n, initialized to all 0s
    stack = empty stack
    SCCs = empty list

    # Step 1: First DFS
    for each vertex u in G:
        if visited[u] == 0:
            DFS(G, u, visited, stack)

    # Step 2: Transpose the graph
    G_T = transpose(G)

    # Step 3: Second DFS
    visited = array of size n, initialized to all 0s
    while stack is not empty:
        u = stack.pop()
        if visited[u] == 0:
            SCC = []
            DFS(G_T, u, visited, SCC)
            SCCs.append(SCC)

    return SCCs

function DFS(G, u, visited, stack):
    visited[u] = 1
    for each vertex v in G[u]:
        if visited[v] == 0:
            DFS(G, v, visited, stack)
    stack.push(u)

function transpose(G):
    G_T = empty graph with the same vertices as G
    for each vertex u in G:
        for each vertex v in G[u]:
            G_T.add_edge(v, u)
    return G_T
```

This algorithm has a time complexity of O(V + E), which is linear in the size of the graph.

Solution of problem 5.

(a) Proof:
To prove that root of the DFS tree T is an articulation point if and only if it has at least two children in T, we will first prove the two directions separately.

Direction 1: If the root of the DFS tree T has at least two children in T, then it is an articulation point.

Let the root of T be denoted by r. If r has at least two children in T, then removing r would disconnect T into at least two components. Specifically, each subtree rooted at one of r's children would be a separate component. Therefore, r is an articulation point.

Direction 2: If the root of the DFS tree T is an articulation point, then it has at least two children in T.

Let the root of T be denoted by r. If r is an articulation point, then removing r would disconnect T into at least two components. Let C1 and C2 be two such components, where C1 contains r and its children, and C2 contains the rest of T. Since r is the root of T, it must be connected to every other vertex in T. Therefore, there must be at least one edge from C2 to C1, which means that there must be at least one child of r in T. Since r is an articulation point, there must be at least one other child of r in T. Therefore, r has at least two children in T.

Combining the two directions, we have proved that the root of the DFS tree T is an articulation point if and only if it has at least two children in T.

(b) Proof:

We will prove the statement using the contrapositive. That is, we will prove that if v is not an articulation point, then it does not have a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

Assume that v is not an articulation point of G. This means that removing v and its incident edges does not disconnect G. Thus, all neighbors of v will still be connected to the rest of the graph.

Now, consider any child s of v in T. If there is a back edge from s or any descendant of s to a proper ancestor of v, then removing v will not disconnect the graph. This is because the back edge will provide an alternative path from s or its descendant to the rest of the graph, bypassing v. Therefore, v cannot have a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

Conversely, assume that v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v. We will prove that v is an articulation point of G.

Removing v from G will disconnect its neighbors from the rest of the graph. Since v is not a leaf node (otherwise it cannot have a child s), removing v will create at least two components in the graph. Let C be the component of G that contains s. Removing v from G will also disconnect C from the rest of the graph, because there is no back edge from s or any of its descendants to a proper ancestor of v. Therefore, v is an articulation point of G.

Hence, we have proven that v is an articulation point of G if and only if it has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

(c) Here is the pseudo-code for finding the articulation points in a connected undirected graph using DFS:

```
procedure DFS(Graph G, Vertex v, integer &time, integer *discovery,
    integer *low, integer *parent, boolean *articulation_points):
children = 0
discovery[v] = time
low[v] = time
```

```
    time = time + 1
    for each vertex w in G.adj[v]:
        if discovery[w] == -1:
            parent[w] = v
            children = children + 1
            DFS(G, w, time, discovery, low, parent, articulation_points)
            low[v] = min(low[v], low[w])
            if parent[v] == -1 and children > 1:
                articulation_points[v] = true
            if parent[v] != -1 and low[w] >= discovery[v]:
                articulation_points[v] = true
        else if w != parent[v]:
            low[v] = min(low[v], discovery[w])

    function find_articulation_points(Graph G):
        n = number of vertices in G
        time = 0
        discovery = array of n integers, initialized to -1
        low = array of n integers, initialized to -1
        parent = array of n integers, initialized to -1
        articulation_points = array of n booleans, initialized to false
        for each vertex v in G:
            if discovery[v] == -1:
                DFS(G, v, time, discovery, low, parent,
                    articulation_points)
        return articulation_points
```

The DFS procedure is the standard DFS algorithm with some additional bookkeeping. discovery[v] is the time at which the vertex v is discovered, low[v] is the lowest time of any vertex w reachable from v by either a tree edge or a back edge, parent[v] is the parent of vertex v in the DFS tree, and articulation points[v] is a boolean indicating whether vertex v is an articulation point.

The find articulation points function initializes the discovery, low, parent, and articulation points arrays, and then runs DFS on each vertex that has not yet been discovered. It returns the articulation points array, which contains a boolean value for each vertex indicating whether it is an articulation point or not.

Solution of problem 6.

(a) One approach to solve this problem is to use a modified version of the depth-first search algorithm. We start by selecting any rebel as the root of our search tree and perform a depth-first search from this root, keeping track of the number of visited nodes.

During the search, we assign each visited node a unique identifier, which represents the order in which we discovered the node. For each node u, we also keep track of the earliest discovered node that is reachable from u, called low(u). This can be done by initializing low(u) to u's discovery time and updating it whenever we discover a node v such that v is not u's parent and low(v) is less than or equal to u's discovery time.

If at any point during the search we find a node u such that low(u) is less than u's discovery time, then u is an articulation point, meaning that removing u from the graph would result in a disconnected graph. In our case, this means that there is no single rebel to whom we can send the message such that it reaches all rebels.

Otherwise, if the root has more than one child in the search tree, then it is the vertex v we are looking for, as sending a message to v will propagate it to all rebels. Otherwise, we repeat the algorithm, selecting a different root for the search tree.

Here is the pseudocode for the algorithm:

```
find_rebel_with_message(n, contacts):
    visited = [False] * n
    low = [None] * n
    discovery_time = [None] * n
    is_articulation = [False] * n
    root = 0
    count = 0

    def dfs(u, parent):
        nonlocal root, count
        visited[u] = True
        count += 1
        discovery_time[u] = count
        low[u] = discovery_time[u]
        child_count = 0
        for v in contacts[u]:
            if not visited[v]:
                child_count += 1
                dfs(v, u)
                low[u] = min(low[u], low[v])
                if low[v] >= discovery_time[u] and parent != -1:
                    is_articulation[u] = True
                if parent == -1 and child_count > 1:
                    root = u
            elif v != parent:
                low[u] = min(low[u], discovery_time[v])

    dfs(root, -1)
    if any(is_articulation):
        return "no solution"
    elif len(contacts[root]) > 1:
        return root
    else:
        for u in range(n):
            if u != root and len(contacts[u]) > 1:
                return u
        return "no solution"
```

The running time of the algorithm is O(n) because we are only visiting each vertex once in the worst case. We are also only performing constant time operations for each vertex, which does not depend on the size of the graph. Therefore, the time complexity of the algorithm is linear in the number of vertices n.

(b) **Algorithm:** To find the minimal number of rebels that need to be sent a direct message before it can be propagated to all rebels, we can use the following algorithm:

Construct a directed graph G with vertices representing the rebels and edges representing the communication channels between them. Perform a depth-first search (DFS) on G to calculate the finishing times for each vertex. Reverse the direction of all edges in G to obtain the transpose graph GT. Perform another DFS on GT, visiting vertices in the order of decreasing finishing times obtained in step 2. Count the number of times a new tree is started during the DFS traversal in step 4. Output the count obtained in step 5 as the minimal number of rebels that need to be sent a direct message.

Pseudocode: The pseudocode for the above algorithm is as follows:

```
function minimalNumberOfMessages(G):
```

```
n = number of vertices in G
visited = [False] * n
finishing_times = []
count = 0
# Step 1: DFS to calculate finishing times
def dfs1(vertex):
    visited[vertex] = True
    for neighbor in G[vertex]:
        if not visited[neighbor]:
            dfs1(neighbor)
    finishing_times.append(vertex)

for vertex in range(n):
    if not visited[vertex]:
        dfs1(vertex)

# Step 2: Create transpose graph GT
GT = [[] for _ in range(n)]
for vertex in range(n):
    for neighbor in G[vertex]:
        GT[neighbor].append(vertex)

# Step 3: DFS on GT to count trees in forest
visited = [False] * n
def dfs2(vertex):
    visited[vertex] = True
    for neighbor in GT[vertex]:
        if not visited[neighbor]:
            dfs2(neighbor)

for vertex in reversed(finishing_times):
    if not visited[vertex]:
        count += 1
        dfs2(vertex)

return count
```

Correctness: This algorithm is correct because it first calculates the finishing times of all vertices in the graph and then uses these finishing times to perform a second DFS on the transpose graph GT in reverse order. By visiting vertices in decreasing order of finishing times, the algorithm ensures that it first visits all vertices in a strongly connected component before visiting any vertex outside that component. Thus, the algorithm accurately identifies the number of strongly connected components in the graph, which is equivalent to the minimal number of rebels that need to be sent a direct message.

Running Time: The time complexity of this algorithm is O(V + E), where V is the number of vertices in the graph and E is the number of edges. This is because the algorithm performs two DFS traversals of the graph, each of which takes O(V + E) time, and the time required to construct the transpose graph GT is O(E).

(c) One possible algorithm to decide if there is a rebel such that if you send them a message, they can propagate it to everyone is as follows:

- Initialize a set of reachable rebels to be empty.

- For each rebel u, perform a BFS starting from u to find all the rebels that can be reached from u, using the groups R(v) as the neighbors of each rebel.
- If the size of the reachable set is n, then output u as the solution.
- If no such u is found in step 3, output ”no solution”.

Pseudocode

```
def has_rebel_to_propagate(n, R):
    for u in range(n):
        reachable = bfs(u, R)
        if len(reachable) == n:
            return u
    return "no solution"

def bfs(u, R):
    reachable = {u}
    queue = [u]
    while queue:
        u = queue.pop(0)
        for v in R[u]:
            if v not in reachable:
                reachable.add(v)
                queue.append(v)
    return reachable
```

The time complexity of this algorithm is O(mn), since we need to consider all the edges between the rebels in the worst case, where m is the total size of the input. Note that we use BFS instead of DFS in this algorithm to ensure that we find the shortest paths to all the reachable rebels.

Solution of problem 7.

To solve the SPACE SUM problem using dynamic programming, we can define an array dp where dp[i] represents the largest sum that can be obtained using the first i elements of A such that at most one of any two consecutive entries is included. We can compute dp[i] as follows:

- If i = 1, then dp[i] = A[1]
- If i = 2, then dp[i] = max(A[1], A[2])
- If i $>$ 2, then dp[i] = max(dp[i-1], dp[i-2] + A[i])

The first two cases are the base cases, where we can only consider the first or first two elements of A. For i ¿ 2, we can either include the i-th element or not. If we include the i-th element, then we cannot include the (i-1)-th element, so we add dp[i-2] to A[i]. If we do not include the i-th element, then we can include the (i-1)-th element, so we take dp[i-1].

The final answer is the maximum value in dp.

For the example A = [3, 5, 8, 4, 9, 16, 14, 6, 13], we have:

dp[1] = 3
dp[2] = 5
dp[3] = 11
dp[4] = 11
dp[5] = 20
dp[6] = 27

dp[7] = 34
dp[8] = 34
dp[9] = 47

Therefore, the largest possible sum is **47**.

Here is the Python code implementing this algorithm:

```
def space_sum(A):
    n = len(A)
    if n == 0:
        return 0
    dp = [0] * n
    dp[0] = A[0]
    if n > 1:
        dp[1] = max(A[0], A[1])
    for i in range(2, n):
        dp[i] = max(dp[i-1], dp[i-2] + A[i])
        print(dp[i])
    return dp[n-1]

A = [3, 5, 8, 4, 9, 16, 14, 6, 13]
print(space_sum(A))  # Output: 47
```

Solution of problem 8.

This problem can be formulated as a variant of the knapsack problem where we want to select a subset of k items (companies) that maximize the total score, subject to a constraint on the total time available (i.e., we cannot attend interviews that overlap in time).

We can solve this problem using dynamic programming. Let S(i, t) be the maximum score that can be obtained by attending interviews with the first i companies, given that we have t units of time available. Then, the optimal solution can be obtained by computing S(k, T), where T is the total time available for interviews.

To compute S(i, t), we can use the following recurrence relation:

S(i, t) = max S(i-1, t), S(i-1, t-ti) + si, if ti ¡= t

where ti is the time required for the i-th interview, and si is the score associated with the i-th company. The first term on the right-hand side corresponds to the case where we skip the i-th interview, while the second term corresponds to the case where we attend the i-th interview.

The optimal solution can then be obtained by tracing back the computed values of S(i, t) to determine which companies were selected.

The time complexity of this algorithm is O(kT), where k is the number of companies and T is the total time available. However, if the time values are large, we can use a compressed DP table that only keeps track of the maximum score achieved so far for each time value, thus reducing the time complexity to O(k).

Pseudo-code

```
# Inputs:
# k: number of companies
# T: total time available
# t[1...k]: array of time values for each company
# s[1...k]: array of score values for each company
```

```
# Initialize DP table
S[0...k, 0...T] = 0

# Compute DP table
for i = 1 to k do
    for j = 1 to T do
        if t[i] <= j then
            S[i, j] = max(S[i-1, j], S[i-1, j-t[i]] + s[i])
        else
            S[i, j] = S[i-1, j]

# Find selected companies
selected = []
i = k
j = T
while i > 0 and j > 0 do
    if S[i, j] > S[i-1, j] then
        selected.append(i)
        j = j - t[i]
    i = i - 1

# Return selected companies and maximum score
return selected, S[k, T]
```

In this pseudocode, S[i, j] represents the maximum score that can be obtained by attending interviews with the first i companies, given that we have j units of time available. The selected array keeps track of the indices of the selected companies, and the maximum score is returned as part of the output.

Solution of problem 9.

To design a DP algorithm for counting the number of ways of making change for a given amount n using coins of denominations $a_1, a_2, \ldots, a_k$, we can use the following recursive formula:

Let $C(i, j)$ be the number of ways to make change for amount i using coins up to a_j.

Then, the base case is $C(0, j) = 1$, i.e., there is one way to make change for zero amount. The recursive formula is:

$$\text{C(i, j)} = \begin{cases} C(i, j-1) + C(i - a_j, j), & \text{if } a_j \leq i \\ C(i, j-1), & \text{if } a_j > i \end{cases}$$

The first case represents the situation where we use at least one a_j coin. In this case, we subtract a_j from the total amount i and look for ways to make change for the remaining amount using coins up to a_j. The second case represents the situation where we don't use any a_j coins, so we look for ways to make change for the total amount i using coins up to a_{j-1}.

The final answer would be $C(n, k)$, i.e., the number of ways to make change for amount n using coins up to a_k. Here's the pseudocode for the dynamic programming algorithm to count the number of ways of making change for a given amount n using coins of denominations a1, a2, ..., ak:

```
function countWays(n, a):
// n: total amount, a: array of coin denominations
k = length(a)
// create a 2D array to store the counts
```

```
C = new 2D array of size (n+1) x (k+1)
// initialize the base cases
for j = 0 to k:
    C[0][j] = 1
// compute the counts using the recursive formula
for i = 1 to n:
    for j = 1 to k:
        if a[j-1] <= i:
            C[i][j] = C[i][j-1] + C[i - a[j-1]][j]
        else:
            C[i][j] = C[i][j-1]
// return the final count
return C[n][k]
```

To argue that this algorithm is correct, we need to show that it counts all ways of making change exactly once. To do this, we will use induction on i.

Base case: $C(0, j) = 1$ for all j, which is correct because there is only one way to make change for zero amount.

Inductive step: Suppose that the algorithm correctly counts all ways of making change for all amounts less than i. We will show that it correctly counts all ways of making change for amount i.

Consider any way of making change for amount i using coins up to a_k. Either this way uses at least one a_k coin, or it does not.

If it does not use any a_k coin, then this way must be counted by $C(i, k-1)$ because it only uses coins up to a_{k-1}.

If it does use at least one a_k coin, then we can remove one a_k coin from the total amount and look for ways to make change for the remaining amount using coins up to a_k. By the inductive hypothesis, the algorithm correctly counts all such ways, and each of them can be extended by adding one a_k coin to get a valid way to make change for the total amount i.

Therefore, the algorithm correctly counts all ways of making change for amount i, and the induction is complete.

Solution of problem 10.

(a) The greedy algorithm that at each step selects an interval that covers the largest number of still-uncovered points does not always solve the problem. Here's an example to illustrate this:

Consider the set of points $1, 2, 3, 4.5, 5, 6, 7$. If we apply the greedy algorithm, we would first select the interval $[1, 2]$, which covers points 1 and 2. Then we would select $[2, 3]$, which covers point 3. The next largest interval that covers an uncovered point is $[4.5, 5]$, which covers only one point. So we select it. Then we select $[5, 6]$, which covers points 5 and 6. Finally, we select $[6, 7]$, which covers point 7. Therefore, the greedy algorithm uses 5 intervals to cover all 7 points.

However, we can cover all the points with just 3 intervals, as follows: $[1, 4]$, $[4, 6]$, and $[6, 7]$.

(b) Here's a greedy algorithm that solves the INTERVAL COVER problem:

1. Set $i = 1$ and $k = 0$.
2. While $i \leq n$, do the following:
   (a) Select the smallest possible interval $[p_i, p_{i+1}]$ that covers p_i.
   (b) Increment i to the largest index j such that $p_j \leq p_{i+1}$.
   (c) Increment k.

3. Return k.

To prove that this algorithm always returns a valid solution, we need to show that it covers all n points and that it uses the minimum number of intervals possible.

First, we will show that the algorithm covers all n points. Consider any point p_i. Let $[p_j, p_{j+1}]$ be the interval selected in step 2a that covers p_i. If p_i is not covered by this interval, then there must be some other interval that covers p_i and is smaller than $[p_j, p_{j+1}]$. But the algorithm always selects the smallest possible interval that covers p_i, so this is a contradiction. Therefore, p_i is covered by $[p_j, p_{j+1}]$.

Next, we will show that the algorithm uses the minimum number of intervals possible. Suppose there is some other solution that uses k' intervals to cover all n points, where $k' < k$. Let $[x_1, x_2]$, $[x_2, x_3]$, ..., $[x_{k'-1}, x_{k'}]$ be the intervals in this solution, in increasing order of their left endpoints. We will show that we can construct a solution using k intervals by modifying this solution.

Consider the leftmost interval $[x_1, x_2]$. Since it covers some point p_i, the greedy algorithm would have selected an interval $[p_i, p_{i+1}]$ with $p_i \leq x_1$. But since $[x_1, x_2]$ is the leftmost interval in the solution, it must cover all points p_i such that $p_i \leq x_1$. Therefore, $[p_i, p_{i+1}]$ must be a subset of $[x_1, x_2]$ for all such p_i. Similarly, we can show that $[x_1, x_2]$ is a subset of $[p_{i'}, p_{i'+1}]$ for some $p_{i'}$ such that $p_{i'} \leq x_2$.

Now, consider the interval $[x_2, x_3]$. Since it covers some point p_j, the greedy algorithm would have selected an interval $[p_j, p_{j+1}]$ with $p_j \leq x_2$. But $[p_{i'}, p_{i'+1}]$ covers all points $p_{i'}$ such that $p_{i'} \leq x_2$, so $[p_j, p_{j+1}]$ must be a subset of $[p_{i'}, p_{i'+1}]$. Similarly, we can show that $[x_2, x_3]$ is a subset of $[p_{i''}, p_{i''+1}]$ for some $p_{i''}$ such that $p_{i''} \leq x_3$. We can repeat this process for all intervals in the solution, and we will end up with k intervals $[p_i, p_{i+1}], [p_{i+1}, p_{i+2}], ..., [p_{i+k-1}, p_{i+k}]$ that cover all n points, where $p_i \leq x_1$ and $p_{i+k} \geq x'_k$. But this means that we can cover all points with k intervals, which contradicts our assumption that the other solution used $k' < k$ intervals. Therefore, the greedy algorithm must use the minimum number of intervals possible.

In conclusion, the greedy algorithm described above always returns a valid solution to the INTERVAL COVER problem, covering all n points with the minimum possible number of intervals.

Solution of problem 11.

Algorithm
This function takes as input the tree data structure and the maximum number of conveyor belts allowed (L), and returns the maximum amount of gold that can be mined using L conveyor belts.

The function uses a nested loop to calculate the maximum amount of gold that can be mined for each node in the tree and for each number of conveyor belts from 1 to L. The maximum amount of gold is computed based on three cases: using a conveyor belt to the parent, not using a conveyor belt to the parent, and using a conveyor belt to one of the children.

Finally, the function finds the maximum amount of gold that can be mined for the whole tree by taking the maximum value of f(root,i)[0] for i = 0 to L. f(u,i)[1] stores source node just before the u.

This algorithm takes time complexity O(N*L*L).

Psedo-code :

```
function calculateMaxGold(tree, L):
// Initialize base cases
for each node u in the tree:
    f(u,0) = [g(u),Null]

// Calculate maximum gold for each node u and i conveyor belts
```

```
for i = 1 to L:
    for each node u in the tree:
        maxGold = f(u,i-1)[0]

        if u has a parent v and f(v,i-1)[1] != u:
            if(maxGold <f(v,i-1)[0] + g(u))):
                    maxGold = f(v,i-1)[0] + g(u)
                    source = u

        if u has a child w  and f(v,i-1)[1] != u:
            for each child w of u:
                if(maxGold <f(w,i-1)[0] + g(u))):
                    maxGold = f(w,i-1)[0] + g(u)
                    source = u

        f(u,i) = [maxGold, source]

// Find the maximum gold for the whole tree
maxTreeGold = 0
for j = 0 to L:
    if maxTreeGold < f(root,j-1)[0]:
        i=j
        while(i>1){
            if f(root,i-1)[1] == root:
                flag =True
                break
            else:
                i-=1
        }
        if flag = True:
            maxTreeGold = f(root,j-1)[0]
return maxTreeGold
```