Algorithms Assignment 3
Jeremy Kwok 100341977

A
Design an algorithm that can identify cycles in a directed graph.

We are going to use a modified DFS. The strategy here is to assign each node a unique ID. DFS keeps track of each node's predecessor. We can use that value to look up each node's predecessor tree. If a node's predecessor tree contains the node iteself, we have found a cycle in the graph, and the cycle is all the nodes (inclusive) between the node and itself in the predecessor tree.

We'll modify the DFS algorithm that we used from class.

```
def DFS(G):
    color = map of V -> COLOR
    p = empty array of |V|
    id = empty array of |V|

    for u in V(G):
        color[u] = WHITE
        p[u] = nil
        id[u] = u

    time = 0
    for u in V(G):
        if color[u] = WHITE:
            DFS_VISIT(u)
        else if color[u] = BLACK:
            checkPredecessors(u)

def DFS_VISIT(u):
    color[u] = GRAY
    for v in Adj[u]:
        if color[v] = WHITE:
            p[v] = u
            DFS_VISIT(v)
        else if color[v] = BLACK:
            checkPredecessors(v)
    color[u] = BLACK

def checkPredecessors(u):
    predList = empty arraylist
    pred = p[u]
    while (pred != nil):
        predList.add(pred)
        if id[pred] = u:
            return predList //Cycle foudn
        pred = p[pred]
```

B

1. Generating sub problems:

For this problem, we are trying to find the solution to problem P(x, coins), that finds lowest amount of coins that form change in the amount of x with coins {c0,c1...cn-1, cn}.

We can say that a subproblem to problem P(x, coins) is finding the lowest amount of coins that form change in the amount of y with coins {c0,c1...cm-1, cm} where y <= x and m <= n.

2. Synthesizing solutions to larger problems

Given the problem P(x, {c0,c1...cm-1, cm}), the solution to this problem is following:
-If x is equal to the value of cm, the solution is 1.
-If x is less than the value of cm, the solution is the solution to the subproblem P(x, {c0,c1...cm-1}).
-If x is greater than the value of cm, the solution is the minimum of the following:
-The solution to the subproblem P(x, {c0,c1...cm-1}). (The same value without considering the highest value coin)
-The solution to the subproblem (1+ P(x - value of cm,{c0,c1...cm})) (1 plus the solution to the problem where we use all the same coins, and we want the value x minus the value of the highest coin)

3. Recursive solution:

```
P(x, {c0,c1...cn}):
    if (x <= 0)
        return 0
    else
        return min(
            P(x - c0, {c0,c1...cn}) + 1,
            P(x - c1, {c0,c1...cn}) + 1,
            ...
            P(x - cn-1 , {c0,c1...cn}) + 1,
            P(x - cn, {c0,c1...cn}) + 1))
```

4. Runtime of the recursive solution:

This function will run through each possible combination of coins for each money value. Therefore at worst case the program has a runtime of each money value ran the number of times equal to the number of different coins.

=Theta(x^n)

5. Bottom up computation (polynomial time)

```
P(x, {c0,c1...cn-1,cn}):
    numCoins = empty_matrix(X,n)
    for i = 1 to n:
        numCoins[1][i] = 1

    for i = 1 to X:
        numCoins[i][1] = c1

    for i = 1 to n:
        for j = 1 to X:
            if (ci > j):
                numCoins[i][j] = numCoins[i-1][j]
            else:
                numCoins[i][j] = min(numCoins[i-1][j], 1 + numCoins[i][j-ci])

    return numCoins[x][n]
```

6. Memoization version of recursive solution (polynomial time)

(M is an empty table x by n large initialized to -1)

```
P(M, x, {c0,c1...cn}):
    if M[x][n] > -1
        return M[x][n]

    if (x <= 0)
        return 0
    else
        q = min(
            P(x - c0, {c0,c1...cn}) + 1,
            P(x - c1, {c0,c1...cn}) + 1,
            ...
            P(x - cn-1 , {c0,c1...cn}) + 1,
            P(x - cn, {c0,c1...cn}) + 1))
        if q < M[x][n]
            M[x][n] = q

    return M[x][n]
```

C

1. Complexity of the Greedy Algorithm

At worst, the greedy algorithm runs though each value once for each coin value. Thus we can say the complexity of the greedy algorithm is at worst Theta(x*n), where n is the number of different coins and x is the value we're trying to reach.

2.

x	Dynamic	Greedy
104	5	5
122	5	5
141	5	5
156	5	5
157	6	6
167	7	7
188	8	8
189	9	9
200	2	2

I have included java code for both of these algorithms.