Camel Banana

total=int(input('Enter no. of bananas at starting:'))

distance=int(input('Enter distance you want to cover:'))

load\_capacity=int(input('Enter max load capacity of your camel:'))

lose=0

start=total

for i in range(distance):

while start>0:

start=start-load\_capacity

if start==1:

lose=lose-1

lose=lose+2

lose=lose-1

start=total-lose

if start==0:

break

print(start)

Traveling Salesman Problem

#include <bits/stdc++.h>   
using namespace std;   
#define V 4

int travllingSalesmanProblem(int graph[][V], int s)

{

vector<int> vertex; for (int i = 0; i < V; i++) if (i != s)

vertex.push\_back(i);

int min\_path = INT\_MAX; do {

int current\_pathweight = 0; int k = s;

for (int i = 0; i < vertex.size(); i++) { current\_pathweight += graph[k][vertex[i]]; k = vertex[i];

}

current\_pathweight += graph[k][s];

min\_path = min(min\_path, current\_pathweight);

}   
while (

next\_permutation(vertex.begin(), vertex.end()));

return min\_path;

}

int main()

{

int graph[][V] = { { 0, 10, 15, 20 },

{ 10, 0, 35, 25 },

{ 15, 35, 0, 30 },

{ 20, 25, 30, 0 } };

int s = 0;

cout << travllingSalesmanProblem(graph, s) << endl;   
return 0;

}

Vacuum Cleaner

def vacuum\_world():

# initializing goal\_state

# 0 indicates Clean and 1 indicates Dirty

goal\_state = {'A': '0', 'B': '0'}

cost = 0

location\_input = input("Enter Location of Vacuum") #user\_input of location vacuum is placed

status\_input = input("Enter status of " + location\_input) #user\_input if location is dirty or clean

status\_input\_complement = input("Enter status of other room")

print("Initial Location Condition" + str(goal\_state))

if location\_input == 'A':

# Location A is Dirty.

print("Vacuum is placed in Location A")

if status\_input == '1':

print("Location A is Dirty.")

# suck the dirt and mark it as clean

goal\_state['A'] = '0'

cost += 1 #cost for suck

print("Cost for CLEANING A " + str(cost))

print("Location A has been Cleaned.")

if status\_input\_complement == '1':

# if B is Dirty

print("Location B is Dirty.")

print("Moving right to the Location B. ")

cost += 1 #cost for moving right

print("COST for moving RIGHT" + str(cost))

# suck the dirt and mark it as clean

goal\_state['B'] = '0'

cost += 1 #cost for suck

print("COST for SUCK " + str(cost))

print("Location B has been Cleaned. ")

else:

print("No action" + str(cost))

# suck and mark clean

print("Location B is already clean.")

if status\_input == '0':

print("Location A is already clean ")

if status\_input\_complement == '1':# if B is Dirty

print("Location B is Dirty.")

print("Moving RIGHT to the Location B. ")

cost += 1 #cost for moving right

print("COST for moving RIGHT " + str(cost))

# suck the dirt and mark it as clean

goal\_state['B'] = '0'

cost += 1 #cost for suck

print("Cost for SUCK" + str(cost))

print("Location B has been Cleaned. ")

else:

print("No action " + str(cost))

print(cost)

# suck and mark clean

print("Location B is already clean.")

else:

print("Vacuum is placed in location B")

# Location B is Dirty.

if status\_input == '1':

print("Location B is Dirty.")

# suck the dirt and mark it as clean

goal\_state['B'] = '0'

cost += 1 # cost for suck

print("COST for CLEANING " + str(cost))

print("Location B has been Cleaned.")

if status\_input\_complement == '1':

# if A is Dirty

print("Location A is Dirty.")

print("Moving LEFT to the Location A. ")

cost += 1 # cost for moving right

print("COST for moving LEFT" + str(cost))

# suck the dirt and mark it as clean

goal\_state['A'] = '0'

cost += 1 # cost for suck

print("COST for SUCK " + str(cost))

print("Location A has been Cleaned.")

else:

print(cost)

# suck and mark clean

print("Location B is already clean.")

if status\_input\_complement == '1': # if A is Dirty

print("Location A is Dirty.")

print("Moving LEFT to the Location A. ")

cost += 1 # cost for moving right

print("COST for moving LEFT " + str(cost))

# suck the dirt and mark it as clean

goal\_state['A'] = '0'

cost += 1 # cost for suck

print("Cost for SUCK " + str(cost))

print("Location A has been Cleaned. ")

else:

print("No action " + str(cost))

# suck and mark clean

print("Location A is already clean.")

# done cleaning

print("GOAL STATE: ")

print(goal\_state)

print("Performance Measurement: " + str(cost))

vacuum\_world()

Cript Arthmentic

#include <bits/stdc++.h>

using namespace std;

vector<int> use(10);

struct node

{

char c;

int v;

};

int check(node\* nodeArr, const int count, string s1,

string s2, string s3)

{

int val1 = 0, val2 = 0, val3 = 0, m = 1, j, i;

for (i = s1.length() - 1; i >= 0; i--)

{

char ch = s1[i];

for (j = 0; j < count; j++)

if (nodeArr[j].c == ch)

break;

val1 += m \* nodeArr[j].v;

m \*= 10;

}

m = 1;

for (i = s2.length() - 1; i >= 0; i--)

{

char ch = s2[i];

for (j = 0; j < count; j++)

if (nodeArr[j].c == ch)

break;

val2 += m \* nodeArr[j].v;

m \*= 10;

}

m = 1;

for (i = s3.length() - 1; i >= 0; i--)

{

char ch = s3[i];

for (j = 0; j < count; j++)

if (nodeArr[j].c == ch)

break;

val3 += m \* nodeArr[j].v;

m \*= 10;

}

if (val3 == (val1 + val2))

return 1;

return 0;

}

bool permutation(const int count, node\* nodeArr, int n,

string s1, string s2, string s3)

{

if (n == count - 1)

{

for (int i = 0; i < 10; i++)

{

if (use[i] == 0)

{

nodeArr[n].v = i;

if (check(nodeArr, count, s1, s2, s3) == 1)

{

cout << "\nSolution found: ";

for (int j = 0; j < count; j++)

cout << " " << nodeArr[j].c << " = "

<< nodeArr[j].v;

return true;

}

}

}

return false;

}

for (int i = 0; i < 10; i++)

{

if (use[i] == 0)

{

nodeArr[n].v = i;

use[i] = 1;

if (permutation(count, nodeArr, n + 1, s1, s2, s3))

return true;

use[i] = 0;

}

}

return false;

}

bool solveCryptographic(string s1, string s2,

string s3)

{

int count = 0;

int l1 = s1.length();

int l2 = s2.length();

int l3 = s3.length();

vector<int> freq(26);

for (int i = 0; i < l1; i++)

++freq[s1[i] - 'A'];

for (int i = 0; i < l2; i++)

++freq[s2[i] - 'A'];

for (int i = 0; i < l3; i++)

++freq[s3[i] - 'A'];

for (int i = 0; i < 26; i++)

if (freq[i] > 0)

count++;

if (count > 10)

{

cout << "Invalid strings";

return 0;

}

node nodeArr[count];

for (int i = 0, j = 0; i < 26; i++)

{

if (freq[i] > 0)

{

nodeArr[j].c = char(i + 'A');

j++;

}

}

return permutation(count, nodeArr, 0, s1, s2, s3);

}

int main()

{

string s1,s2,s3;

cout<<"Enter the First String ";

cin>>s1;

cout<<"Enter the Second String ";

cin>>s2;

cout<<"Enter the Third String ";

cin>>s3;

if (solveCryptographic(s1, s2, s3) == false)

cout << "No solution";

return 0;

}

Graph Coloring

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <set>

using namespace std;

struct Edge {

int src, dest;

};

class Graph

{

public:

vector<vector<int>> adjList;

// Constructor

Graph(vector<Edge> const &edges, int n)

{

adjList.resize(n);

for (Edge edge: edges)

{

int src = edge.src;

int dest = edge.dest;

adjList[src].push\_back(dest);

adjList[dest].push\_back(src);

}

}

};

string color[] =

{

"", "BLUE", "GREEN", "RED", "YELLOW", "ORANGE", "PINK",

"BLACK", "BROWN", "WHITE", "PURPLE", "VOILET"

};

void colorGraph(Graph const &graph, int n)

{

unordered\_map<int, int> result;

for (int u = 0; u < n; u++)

{

set<int> assigned;

for (int i: graph.adjList[u])

{

if (result[i]) {

assigned.insert(result[i]);

}

}

int color = 1;

for (auto &c: assigned )

{

if (color != c) {

break;

}

color++;

}

result[u] = color;

}

for (int v = 0; v < n; v++)

{

cout << "The color assigned to vertex " << v << " is "

<< color[result[v]] << endl;

}

}

int main()

{

vector<Edge> edges = {

{0, 1}, {0, 4}, {0, 5}, {4, 5}, {1, 4}, {1, 3}, {2, 3}, {2, 4}

};

int n = 8;

Graph graph(edges, n);

colorGraph(graph, n);

return 0;

}

BFS

import java.io.\*;

import java.util.\*;

class Main

{

private int V;

private LinkedList<Integer> adj[];

Main(int v)

{

V = v;

adj = new LinkedList[v];

for (int i=0; i<v; i++)

adj[i] = new LinkedList();

}

void addEdge(int v,int w)

{

adj[v].add(w);

}

void BFS(int s)

{

boolean visited[] = new boolean[V];

LinkedList<Integer> queue = new LinkedList<Integer>();

visited[s]=true;

queue.add(s);

while (queue.size() != 0)

{

s = queue.poll();

System.out.print(s+" ");

Iterator<Integer> i = adj[s].listIterator();

while (i.hasNext())

{

int n = i.next();

if (!visited[n])

{

visited[n] = true;

queue.add(n);

}

}

}

}

public static void main(String args[])

{

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

Main g = new Main(n);

while(n>0){

int k = sc.nextInt();

int p = sc.nextInt();

g.addEdge(k,p);

n--;

}

System.out.println("Enter the starting Index");

int s = sc.nextInt();

g.BFS(s);

System.out.println("Following is Breadth First Traversal "+

"(starting from vertex )"+s);

}

}

DFS

import java.io.\*;

import java.util.\*;

class Main {

private int V;

private LinkedList<Integer> adj[];

@SuppressWarnings("unchecked") Main(int v)

{

V = v;

adj = new LinkedList[v];

for (int i = 0; i <v;i++)

adj[i] = new LinkedList();

}

void addEdge(int v, int w)

{

adj[v].add(w);

}

void DFSUtil(int v, boolean visited[])

{

visited[v] = true;

System.out.print(v + " ");

Iterator<Integer> i = adj[v].listIterator();

while (i.hasNext()) {

int n = i.next();

if (!visited[n])

DFSUtil(n, visited);

}

}

void DFS(int v)

{

boolean visited[] = new boolean[V];

DFSUtil(v, visited);

}

public static void main(String args[])

{ Scanner sc = new Scanner(System.in);

System.out.println("Enter Number of Nodes: ");

int n = sc.nextInt();

System.out.println("Enter Number of Relations: ");

int c = sc.nextInt();

Main g = new Main(n);

while(c>0){

int k = sc.nextInt();

int p = sc.nextInt();

g.addEdge(k,p);

c--;

}

System.out.println("Enter the starting Index");

int s = sc.nextInt();

g.DFS(s);

System.out.println("Following is Depth First Traversal "+

"(starting from vertex )"+s);

}

}

Best First Seach

#include <bits/stdc++.h>

using namespace std;

typedef pair<int, int> pi;

vector<vector<pi>> graph;

void addedge(int x, int y, int cost)

{

graph[x].push\_back(make\_pair(cost, y));

graph[y].push\_back(make\_pair(cost, x));

}

void best\_first\_search(int source, int target, int n)

{

vector<bool> visited(n, false);

priority\_queue<pi, vector<pi>, greater<pi>> pq;

pq.push(make\_pair(0, source));

int s = source;

visited[s] = true;

while (!pq.empty())

{

int x = pq.top().second;

cout << x << " ";

pq.pop();

if (x == target)

break;

for (int i = 0; i < graph[x].size(); i++)

{

if (!visited[graph[x][i].second])

{

visited[graph[x][i].second] = true;

pq.push(make\_pair(graph[x][i].first, graph[x][i].second));

}

}

}

}

int main()

{

int v = 14; graph.resize(v);

addedge(0, 1, 3);

addedge(0, 2, 6);

addedge(0, 3, 5);

addedge(1, 4, 9);

addedge(1, 5, 8);

addedge(2, 6, 12);

addedge(2, 7, 14);

addedge(3, 8, 7);

addedge(8, 9, 5);

addedge(8, 10, 6);

addedge(9, 11, 1);

addedge(9, 12, 10);

addedge(9, 13, 2);

int source = 0; int target = 9;

best\_first\_search(source, target, v);

return 0;

}

A\* Algorithm

from collections import deque

class Graph:

def \_init\_(self, adjac\_lis): self.adjac\_lis = adjac\_lis

def get\_neighbors(self, v): return self.adjac\_lis[v]

def h(self, n): H = {

'A': 1,

'B': 1,

'C': 1,

'D': 1

}

return H[n]

def a\_star\_algorithm(self, start, stop):

open\_lst = set([start]) closed\_lst = set([])

poo = {} poo[start] = 0

par = {} par[start] = start

while len(open\_lst) > 0: n = None

for v in open\_lst:

if n == None or poo[v] + self.h(v) < poo[n] + self.h(n): n = v

if n == None:

print('Path does not exist!') return None

if n == stop: reconst\_path = []

while par[n] != n: reconst\_path.append(n) n = par[n]

reconst\_path.append(start)

reconst\_path.reverse()

print('Path found: {}'.format(reconst\_path)) return reconst\_path

for (m, weight) in self.get\_neighbors(n):

if m not in open\_lst and m not in closed\_lst: open\_lst.add(m)

par[m] = n

poo[m] = poo[n] + weight

else:

if poo[m] > poo[n] + weight: poo[m] = poo[n] + weight par[m] = n

if m in closed\_lst: closed\_lst.remove(m) open\_lst.add(m)

open\_lst.remove(n) closed\_lst.add(n)

print('Path does not exist!') return None

adjac\_lis = {

'A': [('B', 1), ('C', 3), ('D', 7)],

'B': [('D', 5)],

'C': [('D', 12)]

}

graph1 = Graph(adjac\_lis)

graph1.a\_star\_algorithm('A', 'D')

Fuzzy Logic

A = dict()

B = dict()

Y = dict()

A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}

B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}

print('The First Fuzzy Set is :', A)

print('The Second Fuzzy Set is :', B)

for A\_key, B\_key in zip(A, B):

A\_value = A[A\_key]

B\_value = B[B\_key]

if A\_value > B\_value:

Y[A\_key] = A\_value

else:

Y[B\_key] = B\_value

print('Fuzzy Set Union is :', Y)

for A\_key, B\_key in zip(A, B):

A\_value = A[A\_key]

B\_value = B[B\_key]

if A\_value < B\_value:

Y[A\_key] = A\_value

else:

Y[B\_key] = B\_value

print('Fuzzy Set Intersection is :', Y)

for A\_key in A:

Y[A\_key]= 1-A[A\_key]

print('Fuzzy Set Complement Of 1st Fuzzy set is :', Y)

for B\_key in B:

Y[B\_key]= 1-B[B\_key]

print('Fuzzy Set Complement Of 2nd Fuzzy set is :', Y)

for A\_key, B\_key in zip(A, B):

A\_value = A[A\_key]

B\_value = B[B\_key]

B\_value = 1 - B\_value

if A\_value < B\_value:

Y[A\_key] = A\_value

else:

Y[B\_key] = B\_value

print('Fuzzy Set Difference is :', Y)

Resolution:

import copy

import time

class Parameter:

variable\_count = 1

def \_\_init\_\_(self, name=None):

if name:

self.type = "Constant"

self.name = name

else:

self.type = "Variable"

self.name = "v" + str(Parameter.variable\_count)

Parameter.variable\_count += 1

def isConstant(self):

return self.type == "Constant"

def unify(self, type\_, name):

self.type = type\_

self.name = name

def \_\_eq\_\_(self, other):

return self.name == other.name

def \_\_str\_\_(self):

return self.name

class Predicate:

def \_\_init\_\_(self, name, params):

self.name = name

self.params = params

def \_\_eq\_\_(self, other):

return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

def \_\_str\_\_(self):

return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

def getNegatedPredicate(self):

return Predicate(negatePredicate(self.name), self.params)

class Sentence:

sentence\_count = 0

def \_\_init\_\_(self, string):

self.sentence\_index = Sentence.sentence\_count

Sentence.sentence\_count += 1

self.predicates = []

self.variable\_map = {}

local = {}

for predicate in string.split("|"):

name = predicate[:predicate.find("(")]

params = []

for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):

if param[0].islower():

if param not in local: # Variable

local[param] = Parameter()

self.variable\_map[local[param].name] = local[param]

new\_param = local[param]

else:

new\_param = Parameter(param)

self.variable\_map[param] = new\_param

params.append(new\_param)

self.predicates.append(Predicate(name, params))

def getPredicates(self):

return [predicate.name for predicate in self.predicates]

def findPredicates(self, name):

return [predicate for predicate in self.predicates if predicate.name == name]

def removePredicate(self, predicate):

self.predicates.remove(predicate)

for key, val in self.variable\_map.items():

if not val:

self.variable\_map.pop(key)

def containsVariable(self):

return any(not param.isConstant() for param in self.variable\_map.values())

def \_\_eq\_\_(self, other):

if len(self.predicates) == 1 and self.predicates[0] == other:

return True

return False

def \_\_str\_\_(self):

return "".join([str(predicate) for predicate in self.predicates])

class KB:

def \_\_init\_\_(self, inputSentences):

self.inputSentences = [x.replace(" ", "") for x in inputSentences]

self.sentences = []

self.sentence\_map = {}

def prepareKB(self):

self.convertSentencesToCNF()

for sentence\_string in self.inputSentences:

sentence = Sentence(sentence\_string)

for predicate in sentence.getPredicates():

self.sentence\_map[predicate] = self.sentence\_map.get(

predicate, []) + [sentence]

def convertSentencesToCNF(self):

for sentenceIdx in range(len(self.inputSentences)):

# Do negation of the Premise and add them as literal

if "=>" in self.inputSentences[sentenceIdx]:

self.inputSentences[sentenceIdx] = negateAntecedent(

self.inputSentences[sentenceIdx])

def askQueries(self, queryList):

results = []

for query in queryList:

negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))

negatedPredicate = negatedQuery.predicates[0]

prev\_sentence\_map = copy.deepcopy(self.sentence\_map)

self.sentence\_map[negatedPredicate.name] = self.sentence\_map.get(

negatedPredicate.name, []) + [negatedQuery]

self.timeLimit = time.time() + 40

try:

result = self.resolve([negatedPredicate], [

False]\*(len(self.inputSentences) + 1))

except:

result = False

self.sentence\_map = prev\_sentence\_map

if result:

results.append("TRUE")

else:

results.append("FALSE")

return results

def resolve(self, queryStack, visited, depth=0):

if time.time() > self.timeLimit:

raise Exception

if queryStack:

query = queryStack.pop(-1)

negatedQuery = query.getNegatedPredicate()

queryPredicateName = negatedQuery.name

if queryPredicateName not in self.sentence\_map:

return False

else:

queryPredicate = negatedQuery

for kb\_sentence in self.sentence\_map[queryPredicateName]:

if not visited[kb\_sentence.sentence\_index]:

for kbPredicate in kb\_sentence.findPredicates(queryPredicateName):

canUnify, substitution = performUnification(

copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

if canUnify:

newSentence = copy.deepcopy(kb\_sentence)

newSentence.removePredicate(kbPredicate)

newQueryStack = copy.deepcopy(queryStack)

if substitution:

for old, new in substitution.items():

if old in newSentence.variable\_map:

parameter = newSentence.variable\_map[old]

newSentence.variable\_map.pop(old)

parameter.unify(

"Variable" if new[0].islower() else "Constant", new)

newSentence.variable\_map[new] = parameter

for predicate in newQueryStack:

for index, param in enumerate(predicate.params):

if param.name in substitution:

new = substitution[param.name]

predicate.params[index].unify(

"Variable" if new[0].islower() else "Constant", new)

for predicate in newSentence.predicates:

newQueryStack.append(predicate)

new\_visited = copy.deepcopy(visited)

if kb\_sentence.containsVariable() and len(kb\_sentence.predicates) > 1:

new\_visited[kb\_sentence.sentence\_index] = True

if self.resolve(newQueryStack, new\_visited, depth + 1):

return True

return False

return True

def performUnification(queryPredicate, kbPredicate):

substitution = {}

if queryPredicate == kbPredicate:

return True, {}

else:

for query, kb in zip(queryPredicate.params, kbPredicate.params):

if query == kb:

continue

if kb.isConstant():

if not query.isConstant():

if query.name not in substitution:

substitution[query.name] = kb.name

elif substitution[query.name] != kb.name:

return False, {}

query.unify("Constant", kb.name)

else:

return False, {}

else:

if not query.isConstant():

if kb.name not in substitution:

substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {}

kb.unify("Variable", query.name)

else:

if kb.name not in substitution:

substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {}

return True, substitution

def negatePredicate(predicate):

return predicate[1:] if predicate[0] == "~" else "~" + predicate

def negateAntecedent(sentence):

antecedent = sentence[:sentence.find("=>")]

premise = []

for predicate in antecedent.split("&"):

premise.append(negatePredicate(predicate))

premise.append(sentence[sentence.find("=>") + 2:])

return "|".join(premise)

def getInput(filename):

with open(filename, "r") as file:

noOfQueries = int(file.readline().strip())

inputQueries = [file.readline().strip() for \_ in range(noOfQueries)]

noOfSentences = int(file.readline().strip())

inputSentences = [file.readline().strip()

for \_ in range(noOfSentences)]

return inputQueries, inputSentences

def printOutput(filename, results):

print(results)

with open(filename, "w") as file:

for line in results:

file.write(line)

file.write("\n")

file.close()

if \_\_name\_\_ == '\_\_main\_\_':

inputQueries\_, inputSentences\_ = getInput('C:/shushrut/studies/SRM University/SEM 6/AI/7-Unification Resolutiion/Resolution/Input/input\_1.txt')

knowledgeBase = KB(inputSentences\_)

knowledgeBase.prepareKB()

results\_ = knowledgeBase.askQueries(inputQueries\_)

printOutput("output.txt", results\_)

Unification:

def get\_index\_comma(string):

index\_list = list()

par\_count = 0

for i in range(len(string)):

if string[i] == ',' and par\_count == 0:

index\_list.append(i)

elif string[i] == '(':

par\_count += 1

elif string[i] == ')':

par\_count -= 1

return index\_list

def is\_variable(expr):

for i in expr:

if i == '(' or i == ')':

return False

return True

def process\_expression(expr):

expr = expr.replace(' ', '')

index = None

for i in range(len(expr)):

if expr[i] == '(':

index = i

break

predicate\_symbol = expr[:index]

expr = expr.replace(predicate\_symbol, '')

expr = expr[1:len(expr) - 1]

arg\_list = list()

indices = get\_index\_comma(expr)

if len(indices) == 0:

arg\_list.append(expr)

else:

arg\_list.append(expr[:indices[0]])

for i, j in zip(indices, indices[1:]):

arg\_list.append(expr[i + 1:j])

arg\_list.append(expr[indices[len(indices) - 1] + 1:])

return predicate\_symbol, arg\_list

def get\_arg\_list(expr):

\_, arg\_list = process\_expression(expr)

flag = True

while flag:

flag = False

for i in arg\_list:

if not is\_variable(i):

flag = True

\_, tmp = process\_expression(i)

for j in tmp:

if j not in arg\_list:

arg\_list.append(j)

arg\_list.remove(i)

return arg\_list

def check\_occurs(var, expr):

arg\_list = get\_arg\_list(expr)

if var in arg\_list:

return True

return False

def unify(expr1, expr2):

if is\_variable(expr1) and is\_variable(expr2):

if expr1 == expr2:

return 'Null'

else:

return False

elif is\_variable(expr1) and not is\_variable(expr2):

if check\_occurs(expr1, expr2):

return False

else:

tmp = str(expr2) + '/' + str(expr1)

return tmp

elif not is\_variable(expr1) and is\_variable(expr2):

if check\_occurs(expr2, expr1):

return False

else:

tmp = str(expr1) + '/' + str(expr2)

return tmp

else:

predicate\_symbol\_1, arg\_list\_1 = process\_expression(expr1)

predicate\_symbol\_2, arg\_list\_2 = process\_expression(expr2)

# Step 2

if predicate\_symbol\_1 != predicate\_symbol\_2:

return False

# Step 3

elif len(arg\_list\_1) != len(arg\_list\_2):

return False

else:

# Step 4: Create substitution list

sub\_list = list()

# Step 5:

for i in range(len(arg\_list\_1)):

tmp = unify(arg\_list\_1[i], arg\_list\_2[i])

if not tmp:

return False

elif tmp == 'Null':

pass

else:

if type(tmp) == list:

for j in tmp:

sub\_list.append(j)

else:

sub\_list.append(tmp)

# Step 6

return sub\_list

if \_\_name\_\_ == '\_\_main\_\_':

f1 = 'Q(a, g(x, a), f(y))'

f2 = 'Q(a, g(f(b), a), x)'

# f1 = input('f1 : ')

# f2 = input('f2 : ')

result = unify(f1, f2)

if not result:

print('The process of Unification failed!')

else:

print('The process of Unification successful!')

print(result)