**AI LAB : EXP1**

Saket Kumar Baranwal(RA1911003010414) – G1

Aim :- To find the maximum number of bananas that can be delivered by camel to the market.

Procedure:-

We have a total of 3000 bananas.

The destination is 1000KMs

Only 1 mode of transport.

Camel can carry a maximum of 1000 banana at a time.

Camel eats a banana every km it travels.

With all these points, we can say that person won’t we able to transfer any banana

to the destination as the camel is going to eat all the banana on its way to the

destination.

But the trick here is to have intermediate drop points, then, the camel can make

several short trips in between.

Also, we try to maintain the number of bananas at each point to be multiple of

1000.

**Code:-**

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

#Here if condition is checking that camel doesn't move back if there is only one banana left.

if start==1:

lose=lose-1#Lose is decreased because if camel try to get remaining one banana he will lose one extra banana for covering that two miles.

#Here we are increasing lose because for moving backward and forward by one mile two bananas will be lose

lose=lose+2

#Here lose is decreased as in last trip camel will not go back.

lose=lose-1

start=total-lose

if start==0:#Condition to check whether it is possible to take a single banana or not.

break

print(start)

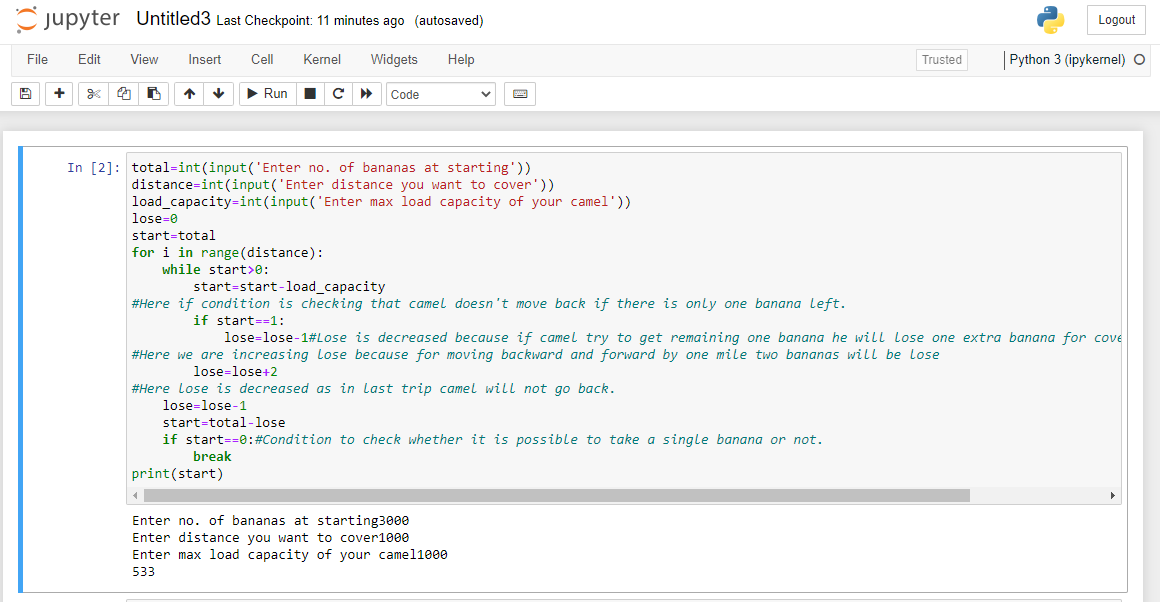
Output:-

Enter no. of bananas at starting:3000

Enter distance you want to cover:1000

Enter max load capacity of your camel:1000

533



**Aim:-** To study Missionaries and Cannibals Problem

Code:-

import math

class State():

    def \_\_init\_\_(self, cannibalLeft, missionaryLeft, boat, cannibalRight, missionaryRight):

    self.cannibalLeft = cannibalLeft

    self.missionaryLeft = missionaryLeft

    self.boat = boat

    self.cannibalRight = cannibalRight

    self.missionaryRight = missionaryRight

    self.parent = None

    def is\_goal(self):

    if self.cannibalLeft == 0 and self.missionaryLeft == 0:

    return True

    else:

    return False

    def is\_valid(self):

    if self.missionaryLeft >= 0 and self.missionaryRight >= 0 \

                and self.cannibalLeft >= 0 and self.cannibalRight >= 0 \

                and (self.missionaryLeft == 0 or self.missionaryLeft >= self.cannibalLeft) \

                and (self.missionaryRight == 0 or self.missionaryRight >= self.cannibalRight):

    return True

    else:

    return False

    def \_\_eq\_\_(self, other):

    return self.cannibalLeft == other.cannibalLeft and self.missionaryLeft == other.missionaryLeft \

                and self.boat == other.boat and self.cannibalRight == other.cannibalRight \

                and self.missionaryRight == other.missionaryRight

    def \_\_hash\_\_(self):

    return hash((self.cannibalLeft, self.missionaryLeft, self.boat, self.cannibalRight, self.missionaryRight))

def successors(cur\_state):

    children = [];

    if cur\_state.boat == 'left':

    new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 2, 'right',

                               cur\_state.cannibalRight, cur\_state.missionaryRight + 2)

    if new\_state.is\_valid():

    new\_state.parent = cur\_state

    children.append(new\_state)

    new\_state = State(cur\_state.cannibalLeft - 2, cur\_state.missionaryLeft, 'right',

                               cur\_state.cannibalRight + 2, cur\_state.missionaryRight)

    if new\_state.is\_valid():

    new\_state.parent = cur\_state

    children.append(new\_state)

    new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft - 1, 'right',

                               cur\_state.cannibalRight + 1, cur\_state.missionaryRight + 1)

    if new\_state.is\_valid():

    new\_state.parent = cur\_state

    children.append(new\_state)

    new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 1, 'right',

                               cur\_state.cannibalRight, cur\_state.missionaryRight + 1)

    if new\_state.is\_valid():

    new\_state.parent = cur\_state

    children.append(new\_state)

    new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft, 'right',

                               cur\_state.cannibalRight + 1, cur\_state.missionaryRight)

    if new\_state.is\_valid():

    new\_state.parent = cur\_state

    children.append(new\_state)

    else:

    new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 2, 'left',

                               cur\_state.cannibalRight, cur\_state.missionaryRight - 2)

    if new\_state.is\_valid():

    new\_state.parent = cur\_state

    children.append(new\_state)

    new\_state = State(cur\_state.cannibalLeft + 2, cur\_state.missionaryLeft, 'left',

                               cur\_state.cannibalRight - 2, cur\_state.missionaryRight)

    if new\_state.is\_valid():

    new\_state.parent = cur\_state

    children.append(new\_state)

    new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft + 1, 'left',

                               cur\_state.cannibalRight - 1, cur\_state.missionaryRight - 1)

    if new\_state.is\_valid():

    new\_state.parent = cur\_state

    children.append(new\_state)

    new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 1, 'left',

                               cur\_state.cannibalRight, cur\_state.missionaryRight - 1)

    if new\_state.is\_valid():

    new\_state.parent = cur\_state

    children.append(new\_state)

    new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft, 'left',

                               cur\_state.cannibalRight - 1, cur\_state.missionaryRight)

    if new\_state.is\_valid():

    new\_state.parent = cur\_state

    children.append(new\_state)

    return children

def breadth\_first\_search():

    initial\_state = State(3,3,'left',0,0)

    if initial\_state.is\_goal():

    return initial\_state

    frontier = list()

    explored = set()

    frontier.append(initial\_state)

    while frontier:

    state = frontier.pop(0)

    if state.is\_goal():

    return state

    explored.add(state)

    children = successors(state)

    for child in children:

    if (child not in explored) or (child not in frontier):

    frontier.append(child)

    return None

def print\_solution(solution):

    path = []

    path.append(solution)

    parent = solution.parent

    while parent:

    path.append(parent)

    parent = parent.parent

    for t in range(len(path)):

    state = path[len(path) - t - 1]

    print (" " + str(state.cannibalLeft) + "," + str(state.missionaryLeft) \

                           + "," + state.boat + "," + str(state.cannibalRight) + "," + \

                           str(state.missionaryRight) + " ")

def main():

    solution = breadth\_first\_search()

    print ("Missionaries and Cannibals solution:")

    print ("(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)")

    print\_solution(solution)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:-**

Missionaries and Cannibals solution:

(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)

3,3,left,0,0

1,3,right,2,0

2,3,left,1,0

0,3,right,3,0

1,3,left,2,0

1,1,right,2,2

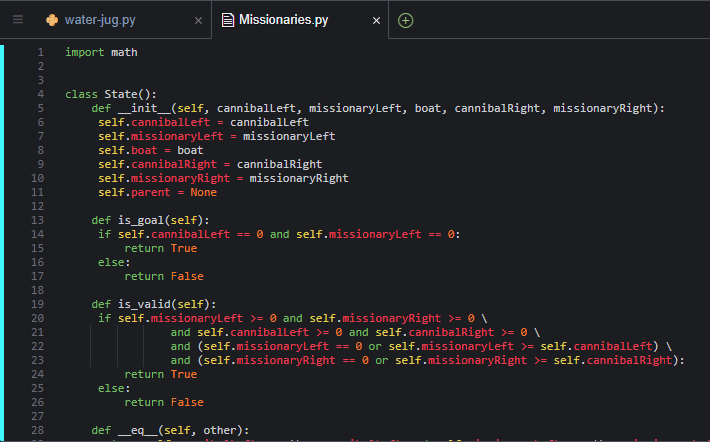
2,2,left,1,1

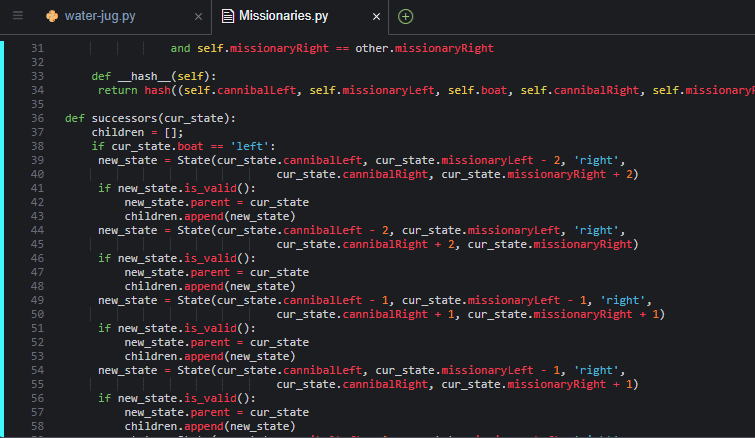
2,0,right,1,3

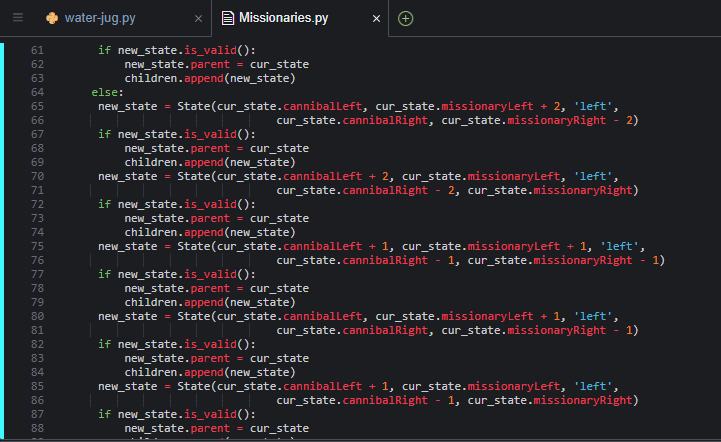
3,0,left,0,3

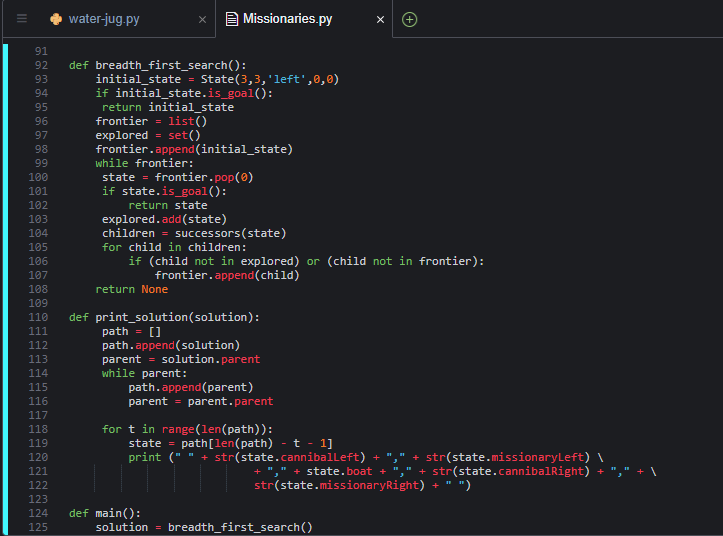
1,0,right,2,3

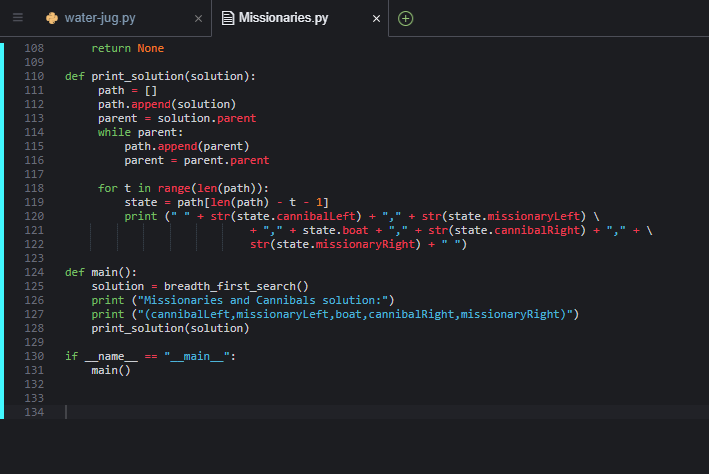
1,1,left,2,2











**Output Screenshot:-**

