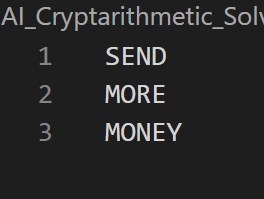
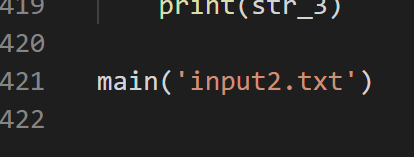
Jia Cheng Li

1) Make an input.txt with words in this format:



-A four letter word + four letter word = 5 letter word

2) Write the name of the input file in line 421:



-In the above example, the input file is called input2.txt.

-Replace the ‘input2.txt’ with the name of your input file follow by the ‘.txt’

3) Run the Python file

Prerequisite: Python and a code compiler

How to run using VSCode:

Capture

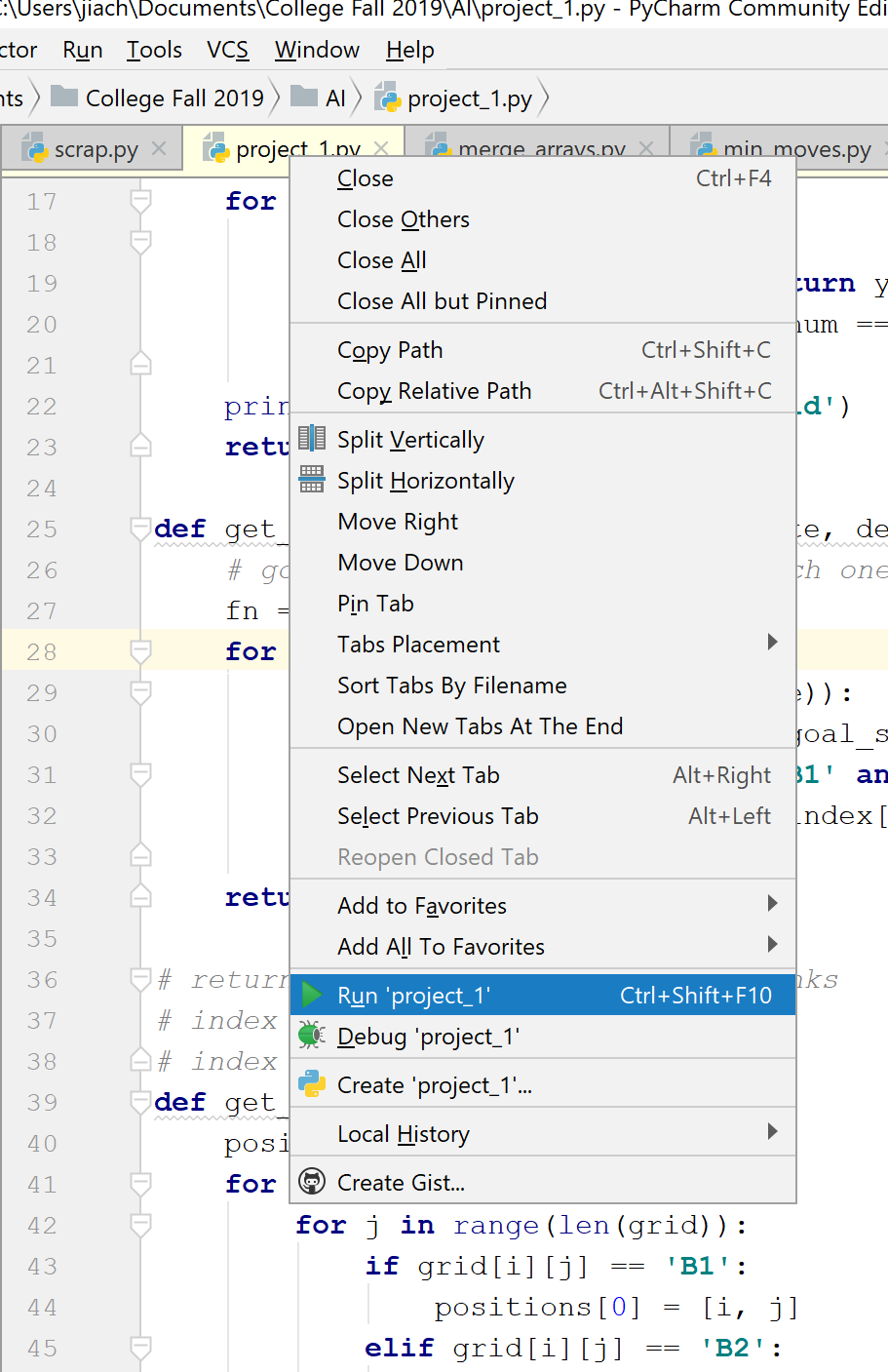
-Go to the directory containing the source code file; in my case it is …..\AI

-Type: python <name of the source code file> in terminal

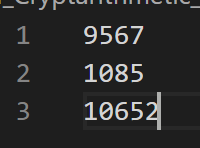
-In my case the name is project\_1.py

How to run using PyCharm:

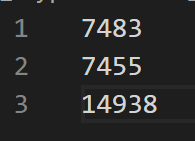
-Go to the file and press the run button in PyCharm:



Output1.txt:



Output2.txt:



Source Code:

import copy

# keeps track of the positions (1-13) and domain(0-9) of a character (e.g. x1)

class Node():

    def \_\_init\_\_(self, char):

        self.char = char

        self.positions = []

        self.domain = []

# help keep track of the current assigned values and carry overs

class State():

    def \_\_init\_\_(self):

        self.values = []

        self.assigned = set()

        self.C1 = None

        self.C2 = None

        self.C3 = None

        self.C4 = None

# make a dict with the position as the key from a character\_dict (char as key)

def make\_position\_dict(char\_dict):

    postion\_dict = {}

    for key, val in char\_dict.items():

        for i in val.positions:

            if i not in postion\_dict:

                postion\_dict[i] = val.char

    return postion\_dict

# define the appropriate domain for each char

def assign\_domain(s1, s2, s3):

    print(s1, s2, s3)

    print(len(s1), len(s2), len(s3))

    char\_dict = {}

    # make a dict for each char containing its positions

    str\_lst = [s1, s2, s3]

    counter = 1

    for word in str\_lst:

        for i in word:

            if i not in char\_dict:

                char\_dict[i] = Node(i)

            char\_dict[i].positions.append(counter)

            counter += 1

    # get domains for each char

    for char, node in char\_dict.items():

        # only domain for x9 is 1

        if 9 in node.positions:

            node.domain.append(1)

        else:

            start\_domain = 0

            # 1 and 5 are leading terms, so domain: 1-9

            if 1 in node.positions or 5 in node.positions:

                start\_domain = 1 # start at 1

            for i in range(start\_domain, 10):

                node.domain.append(i)

    # make position dict

    position\_dict = make\_position\_dict(char\_dict)

    # assign edge case domains

    edge\_positions = [10, 13]

    for num in edge\_positions:

        # if previous 2 terms are same char then their sum gives even domain

        if position\_dict[num-5] == position\_dict[num-9]:

            char = position\_dict[num]

            char\_dict[char].domain = [0, 2, 4, 6, 8]

    return char\_dict

# true if val is not already assigned in state

# check if the number works out in the addition

# if work then use this func again with modify = 1 to modify state

def can\_add\_val(state, val, char\_dict, char, modify):

    # False if val is already used by a letter

    if val in state.values:

        return False

    positions = char\_dict[char].positions

    values = copy.deepcopy(state.values)

    C1, C2, C3, C4 = state.C1, state.C2, state.C3, state.C4

    for p in positions:

        values[p-1] = val

    x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13 = \

    values[0], values[1], values[2], values[3], values[4], values[5], \

    values[6], values[7], values[8], values[9], values[10],\

    values[11], values[12]

    if x4 != -1 and x8 != -1 and x13 != -1:

        val = x4 + x8

        if val >= 10:

            C1 = 1

        else:

            C1 = 0

        if x4 + x8 != x13 + C1\*10:

            return False

    if x3 != -1 and x7 != -1 and x12 != -1:

        # if C1 is not calculated then test for C1 = 0 and C1 = 1

        # return False if fail for both

        if C1 == None:

            # test when c1 == 0

            # val = x3 + x7 +0

            flag = True

            C1 = 0

            val = x3 + x7 + C1

            if val >= 10:

                C2 = 1

            else:

                C2 = 0

            if x3 + x7 + C1 != x12 + C2\*10:

                flag = False

            if not flag:

                # test when c1 == 1

                C1 = 1

                val = x3 + x7 + C1

                if val >= 10:

                    C2 = 1

                else:

                    C2 = 0

                if x3 + x7 + C1 != x12 + C2\*10:

                    return False

        else:

            val = x3 + x7 + C1

            if val >= 10:

                C2 = 1

            else:

                C2 = 0

            if x3 + x7 + C1 != x12 + C2\*10:

                return False

    if x2 != -1 and x6 != -1 and x11 != -1:

        # if C1 is not calculated then test for C1 = 0 and C1 = 1

        # return False if fail for both

        if C2 == None:

            flag = True

            C2 = 0

            val = x2 + x6 + C2

            if val >= 10:

                C3 = 1

            else:

                C3 = 0

            if x2 + x6 + C2 != x11 + C3\*10:

                flag = False

            if not flag:

                C2 = 1

                val = x2 + x6 + C2

                if val >= 10:

                    C3 = 1

                else:

                    C3 = 0

                if x2 + x6 + C2 != x11 + C3\*10:

                    return False

        else:

            val = x2 + x6 + C2

            if val >= 10:

                C3 = 1

            else:

                C3 = 0

            if x2 + x6 + C2 != x11 + C3\*10:

                return False

    if x1 != -1 and x5 != -1 and x10 != -1:

        # if C1 is not calculated then test for C1 = 0 and C1 = 1

        # return False if fail for both

        if C3 == None:

            flag = True

            C3 = 0

            val = x1 + x5 + C3

            if val >= 10:

                C4 = 1

            else:

                C4 = 0

            if x1 + x5 + C3 != x10 + C4\*10:

                flag = False

            if not flag:

                C3 = 1

                val = x1 + x5 + C3

                if val >= 10:

                    C4 = 1

                else:

                    C4 = 0

                if x1 + x5 + C3 != x10 + C4\*10:

                    return False

        else:

            val = x1 + x5 + C3

            if val >= 10:

                C4 = 1

            else:

                C4 = 0

            if x1 + x5 + C3 != x10 + C4\*10:

                return False

        # C4 == x9 == 1

        if C4 == 0:

            return False

    if modify:

        state.values = values

        print('values')

        print(state.values)

        # modify values in state

        state.C1, state.C2, state.C3, state.C4 = C1, C2, C3, C4

        print('CHAR:', char)

        # add char to assigned

        state.assigned.add(char)

        print('assigned')

        print(state.assigned)

    return True

# iterate through the positions where the character is located and

# return the lowest number of empty neighbors

def get\_empty\_neighbors(char, state, char\_dict):

    min\_num = None

    # check empty neighbors for each position

    positions = char\_dict[char].positions

    for i in positions: # positions from 1-13

        num\_empty = 0

        # current position in state: state.values[i-1]

        if i - 1 > 0:

            if state.values[i-2] == -1:

                num\_empty += 1

        if i + 1 < 14:

            if state.values[i] == -1:

                num\_empty += 1

        if min\_num == None:

            min\_num = num\_empty

        elif num\_empty < min\_num:

            min\_num = num\_empty

    return min\_num

# returns the char with the lowest domain

# if tie, then go with the char with the lowest unassigned neighbors

# if tie again, then choose a random tied char

def select\_var(char\_dict, state):

    min\_char = None

    tied\_domains = set()

    for char, node in char\_dict.items():

        if char in state.assigned:

            continue

        if min\_char == None:

            min\_char = char

        else:

            min\_len = len(char\_dict[min\_char].domain)

            curr\_len = len(char\_dict[char].domain)

            # add any tied chars

            if curr\_len == min\_len:

                tied\_domains.add(min\_char)

                tied\_domains.add(char)

            elif curr\_len < min\_len:

                min\_char = char

    # domains tied- choose the key with the fewest unassigned neighbors

    # if tied- just return the current lowest one

    if len(tied\_domains) > 0:

        min\_char = None

        min\_num\_empty = None

        for i in tied\_domains:

            if min\_char == None:

                min\_char = i

                min\_num\_empty = get\_empty\_neighbors(min\_char, state, char\_dict)

            else:

                curr\_empty = get\_empty\_neighbors(i, state, char\_dict)

                if curr\_empty < min\_num\_empty:

                    min\_char = i

                    min\_num\_empty = curr\_empty

    return min\_char

def copy\_state(state):

    res\_state = State()

    res\_state.values= state.values

    res\_state.C1 = state.C1

    res\_state.C2 = state.C2

    res\_state.C3 = state.C3

    res\_state.C4 = state.C4

    return res\_state

def copy\_dict(dict\_to\_copy):

    res\_dict = {}

    for key, val in dict\_to\_copy.items():

        if key not in res\_dict:

            res\_dict[key] = Node(val.char)

            # deep copy of positions

            res\_positions = []

            for i in val.positions:

                res\_positions.append(i)

            res\_dict[key].positions = res\_positions

            # deep copy of domain

            res\_domain = []

            for i in val.domain:

                res\_domain.append(i)

            res\_dict[key].domain = res\_domain

    return res\_dict

def check(state):

    values = state.values

    x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13 = values[0],  \

    values[1], values[2], values[3], values[4], values[5], values[6], values[7], \

    values[8], values[9], values[10], values[11], values[12]

    C1, C2, C3, C4 = 0, 0, 0, 0

    if x13 >= 10:

        C1 = 1

        # x13 = 10- x13

    if x12 >= 10:

        C2 = 1

        # x12 = 10- x12

    if x11 >= 10:

        C3 = 1

        # x11 = 10- x11

    if x10 >= 10:

        C4 = 1

        # x10 = 10- x10

    # constraints

    # x4 + x8 = x13 + C1\*10

    # x3 + x7 + C1 = x12 + C2\*10

    # x2 + x6 + C2 = x11 + C3\*10

    # x1 + x5 + C3 = x10 + C4\*10

    # C4 = x9

    if (x4 + x8 == x13 + C1\*10) and (x3 + x7 + C1 == x12 + C2\*10) \

    and (x2 + x6 + C2 == x11 + C3\*10) and (x1 + x5 + C3 == x10 + C4\*10) and (C4 == x9):

        return True

    return False

# done if positions are all assigned and values are all uniq

def done(state):

    values = state.values

    # all values have to be assigned, e.g. not = -1

    for i in values:

        if i == -1:

            return False

    # check if satisfy constraint

    return check(state)

# def remove\_position(position\_dict, position):

#     new\_dict = {}

#     for key, val in position\_dict.items():

#         if key != position:

#             new\_dict[key] = val

#     return new\_dict

# change modify values back

def remove\_assigned(state, char, positions):

    state.assigned.remove(char)

    # reset the values to -1

    for i in positions:

        # reset any carry over to none

        if i in [4, 8, 13]:

            if i == 13: state.C1 = None

            state.C1 = None

        elif i in [3, 7, 12]:

            if i == 12: state.C2 = None

            state.C2 = None

        elif i in [2, 6, 11]:

            if i == 11: state.C3 = None

            state.C3 = None

        elif i in [1, 5, 10]:

            if i == 10: state.C4 = None

            state.C4 = None

        state.values[i-1] = -1

def backtrack(char\_dict, state):

    print("-----------------------")

    values = state.values

    # check if done

    flag = True

    # check if all varaibles have a unique number

    for i in values:

        if i == -1:

            flag = False

            break

    if flag: # if all variables are filled then check constraints

        return check(state)

    # select unassigned var

    char = select\_var(char\_dict, state)

    print(char, char\_dict[char].positions, char\_dict[char].domain)

    # get domain val from var (can make var a node)

    domain\_vals = char\_dict[char].domain

    for i in domain\_vals:

        if can\_add\_val(state, i, char\_dict, char, 0):

            print('domain\_vals', domain\_vals)

            print('domain:', i)

            can\_add\_val(state, i, char\_dict, char, 1) # apply changes to res\_state

            backtrack(char\_dict, state)

            # check if done

            flag = True

            for i in state.values:

                if i == -1:

                    flag = False

                    break

            # if all are assigned then should be correct since

            # only add numbers that satisfy contraints

            if flag: return True

            positions = char\_dict[char].positions

            remove\_assigned(state, char, positions)

    print('failed')

    return False

def main(file\_name):

    with open(file\_name) as f:

        # get words from files

        words = f.readlines()

        # get rid of any \n

        for i in range(len(words)):

            words[i] = words[i].strip()

    # assign each word a domain of nums

    char\_dict = assign\_domain(words[0], words[1], words[2])

    for key, node in char\_dict.items():

        print(key, node.char, node.positions, node.domain)

    state = State()

    initial\_values = []

    for i in range(13):

        initial\_values.append(-1)

    state.values = initial\_values

    res = backtrack(char\_dict, state)

    print(res)

    print(state.assigned)

    values = state.values

    str\_1 = ''

    str\_2 = ''

    str\_3 = ''

    for i in values[0:4]: str\_1 += str(i)

    for i in values[4:8]: str\_2 += str(i)

    for i in values[8:13]: str\_3 += str(i)

    print(str\_1)

    print(str\_2)

    print(str\_3)

main('input2.txt')