**Name: Prashant Kumar**

**Roll No: 222CS2097**

**Semester: 1st (MTech)**

**Q.18. String Matching**

**//stringMatching.py**

# String Pattern Matching KMP Algorithm including regex pattern

import argparse

def match\_pattern(text, pattern):

    positions = []

    text\_length = len(text)

    pattern\_length = len(pattern)

    position = 0

    for i, c in enumerate(text):

        if position < text\_length - pattern\_length and c == pattern[position]:

            offset = 0

            while(offset < pattern\_length):

                if text[i + offset] == pattern[position + offset]:

                    offset = offset + 1

                else:

                    break

            if offset == pattern\_length:

                positions.append(i)

            position = 0

    return positions

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

    parser = argparse.ArgumentParser(description='Brute Force Pattern Matching')

    parser.add\_argument('--case\_sensitive', default = False, type=bool, help='Allow case sensitive checking')

    args = parser.parse\_args()

    case\_sensitive = args.case\_sensitive

    with open('PM\_input.txt') as file:

        lines = file.readlines()

    text = ""

    text = text.join(lines)

    pattern = input('Enter pattern for checking\n')

    print("Text Length : {}".format(len(pattern)))

    print("Text Text : {}".format(len(text)))

    if not case\_sensitive:

        text = text.lower()

        pattern = pattern.lower()

    res = pattern.find("\*")

    if res > 0 :

        print("Initiating Regex Search")

        parts = pattern.split("\*")

        first\_position = match\_pattern(text, parts[0])

        second\_position = match\_pattern(text, parts[1])

        if len(first\_position)>0 or len(second\_position)>0:

            for u in first\_position:

                for v in second\_position:

                    if v >= u + len(parts[0]):

                        print("Matched found at {} {}".format(u,v))

        else:

            print("Pattern not found")

    else:

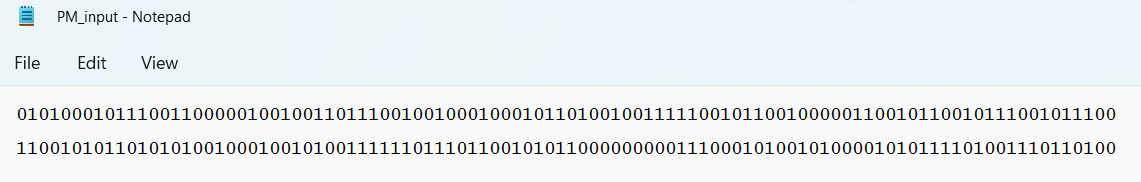
        print("Initiating Normal Search")

        position = match\_pattern(text, pattern)

        for p in position:

            print("Matched found at {}".format(p))

**PM\_input.txt**

****

### Sample output:

Enter pattern for checking

00110

Text Length : 5

Text Text : 100

Initiating Normal Search

Matched found at 12

Matched found at 25

Matched found at 74

Enter pattern for checking

1010\*01101

Text Length : 10

Text Text : 100

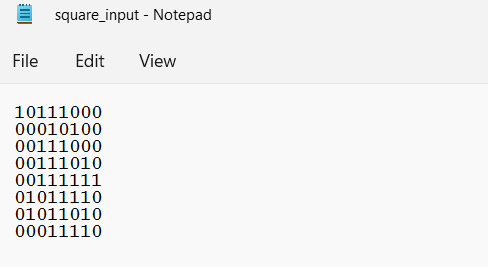
Initiating Regex Search

Matched found at 40 54

Matched found at 46 54

**Q.19. Pattern matching**

**#square\_input.txt**

****

**#SquarePatternMatchingAlgorithm.py**

**Code:**

import numpy as np

class Square\_Pattern:

    def \_\_init\_\_(self) -> None:

        pass

    def get\_dimension(self, array):

        row\_length = len(array)

        col\_length = len(array[0])

        return row\_length, col\_length

    def display\_position(self, positions):

            largest = 0

            for position in positions:

                print("Found Square at {} ofsize {}".format(position['position'], position['size']))

                if int(position['size']) > largest:

                    largest = position['size']

                    # print("Largest Square size is : {}".format(largest))

            return largest

    def show\_largest(self, positions, largest):

        for position in positions:

            if position['size'] == largest:

                print("Found Largest Square at {} ofsize {}".format(position['position'], position['size']))

    def check\_square(self, array):

        # print(array)

        all\_one = True

        length = len(array)

        for m in range(length):

            for n in range(length):

                if array[m][n] != "1":

                    all\_one = False

                    break

            if not all\_one:

                break

        return all\_one

    def find\_square(self, array, rows, cols):

        position = []

        for i in range(rows):

            for j in range(cols):

                e = 2

                if array[i][j] == "1":

                    while (i+e <= rows) and (j+e <= cols):

                         # print("{} <= {} and {} <= {}".format(i+e,rows,j+e,cols))

                         if self.check\_square(array[i:i+e,j:j+e]):

                             position.append({'position' : "({}, {})".format(i, j), 'size' : e})

                             e = e + 1

                         else:

                             break

                    e = 2

                else:

                     continue

                return position

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

    array = []

    with open('square\_input.txt','rb') as file:

        lines = file.readlines()

        print(lines)

        for line in lines:

            col = []

            for c in line.strip():

                col.append(c)

                array.append(col)

                array=np.array(array)

                matcher = Square\_Pattern()

                rows, cols = matcher.get\_dimension(array)

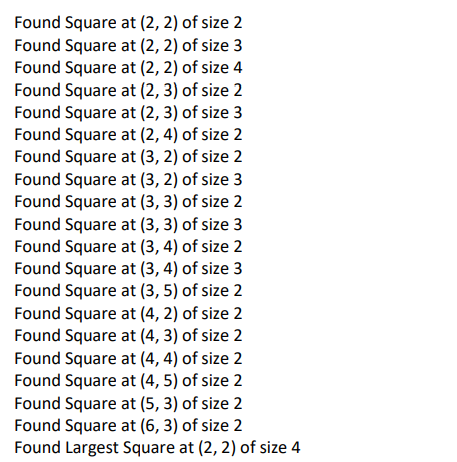
                if rows == cols:

                    positions = matcher.find\_square(array, rows, cols)

                    largest = matcher.display\_position(positions)

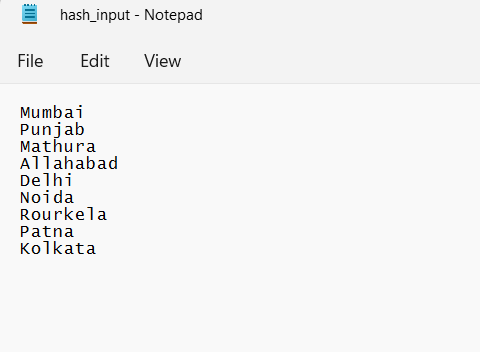
                    matcher.show\_largest(positions, largest)

Output:



**Q. 20. Hash Table**

#hash\_input.txt



#HashTable.py

import random, math

class SymblTable:

    def \_\_init\_\_(self, table\_size):

        self.table\_size = table\_size

        self.HashTable = [[] for \_ in range(table\_size)]

    def display\_hash(self):

        for i in range(len(self.HashTable)):

            print(i, end = " ")

            for j in self.HashTable[i]:

                print("-->", end = " ")

                print(j, end = " ")

            print()

    def Hashing(self, keyvalue):

        k = (math.sqrt(5)-1)/2

        fraction, \_ = math.modf(k\*keyvalue)

        hashvalue = math.floor(self.table\_size \* fraction)

        return hashvalue

    def insert(self, keyvalue, value):

        hash\_key = self.Hashing(keyvalue)

        self.HashTable[hash\_key].append(value)

with open('hash\_input.txt','r') as file:

    lines = file.readlines()

    #print(lines)

    text = ''.join(lines)

    #print(text)

    tokens = text.split()

size = (input('Enter Size of Hash Table\n'))

hash = SymblTable(int(size))

modified = []

for token in tokens:

    if len(token)>10:

        modified.append(token[:10])

    elif len(token)<10:

        extra = ''.join(random.choices('\*', k = 10-len(token)))

        modified.append(token + extra)

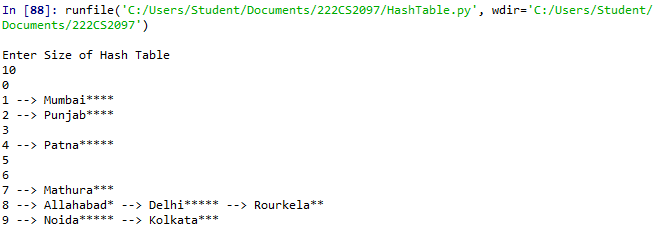
for token in modified:

    ascii\_sum = sum([ord(c) - 96 for c in token])

    hash.insert(ascii\_sum, token)

hash.display\_hash()

Output:



**=========================End=========================**