**Name: Prashant Kumar**

**Roll No: 222CS2097**

**Semester: 1st (MTech)**

**Q.18. String Matching**

**//stringMatching.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

char \*randstr ( int l )

{

   char \*S;

   int i;

   S = (char \*)malloc((l+1) \* sizeof(char));

   S[l] = '\0';

   for (i=0; i<l; ++i) S[i] = '0' + (char)(rand() & 1);

   return S;

}

int \*KMP ( char \*S, char \*T, int n, int m )

{

   char \*TS;

   int i, j, \*match, \*F, \*G, cnt;

   /\* First concatenate T and S to TS \*/

   TS = (char \*)malloc((n+m)\*sizeof(char));

   for (i=0; i<m; ++i) TS[i] = T[i];

   for (i=0; i<n; ++i) TS[m+i] = S[i];

   /\* Then compute the failure function for TS in the usual way \*/

   F = (int \*)malloc((n+m)\*sizeof(int));

   F[0] = 0; i = 1; j = 0;

   while (i < m+n) {

      while ((i < m+n) && (TS[i] == TS[j])) F[i++] = ++j;

      if (j == 0) F[i++] = 0; else j = F[j-1];

   }

   G = (int \*)malloc((m+n)\*sizeof(int));

   for (i=0; i<=m-2; ++i) G[i] = 0;  /\* T cannot be a border in a string of length < m \*/

   G[m-1] = 1;                       /\* T is definitely a border of T \*/

   cnt = 0;                          /\* cnt stores the number of matches found so far \*/

   for (i=m; i<m+n; ++i) {           /\* T is a proper border for larger strings \*/

      if (F[i] < m) G[i] = 0;        /\* The longest proper border is shorter than T \*/

      else if (F[i] == m) G[i] = 1;  /\* T is the longest proper border \*/

      else G[i] = G[F[i]-1];         /\* Check whether T is a border of the longest proper border \*/

      if ((i >= 2\*m - 1) && (G[i])) ++cnt; /\* Match found \*/

   }

   match = (int \*)malloc((cnt+1)\*sizeof(int));

   match[0] = cnt; /\* Store the number of matches in match[0].

                      match[1], match[2], ..., match[cnt] will store the matching indices. \*/

   j = 1;          /\* j is now used as the writing location in match[] \*/

   for (i=2\*m-1; i<m+n; ++i) if (G[i]) match[j++] = i - 2\*m + 1;

   /\* Clean up locally allocated memory \*/

   free(TS); free(F); free(G);

   /\* Return the list of matching indices followed by the count of matches \*/

   return match;

}

/\* This is the function that does string matching based upon the modified KMP

   algorithm implemented abobe. \*/

void strmatch ( int n, int m )

{

   char \*S, \*T;

   int \*match, i;

   S = randstr(n); T = randstr(m);

   printf("S = %s\n", S);

   printf("T = %s\n", T);

   match = KMP(S,T,n,m);

   printf("%d matches found at indices", match[0]);

   for (i=1; i<=match[0]; ++i) printf(" %d", match[i]);

   printf("\n");

   free(S); free(T); free(match);

}

/\* This is the function that does string matching based upon the modified KMP

   algorithm implemented abobe. \*/

void patmatch ( int n , int m )

{

   char \*S, \*T1, \*T2;

   int m1, m2, \*match1, \*match2;

   int i, j, k;

   /\* Generate S, T1 and T2 randomly \*/

   S = randstr(n);

   m1 = m - 1; T1 = randstr(m1);

   m2 = m; T2 = randstr(m2);

   printf("S = %s\n", S);

   printf("T = %s\*%s\n", T1, T2);

   /\* Find all matches of T1 and T2 in S \*/

   match1 = KMP(S,T1,n,m1);

   match2 = KMP(S,T2,n,m2);

   printf("Pattern matches at index pairs");

   for (i=j=1; i<=match1[0]; ++i) { /\* For each match of T1 in S \*/

      k = match1[i] + m1;           /\* Match for T2 in S cannot start before index k \*/

      if (k > match2[match2[0]]) break; /\* No further match possible \*/

      /\* Find the first location of a match \*/

      while ((j <= match2[0]) && (k > match2[j])) ++j;

      /\* Each following match of T2 in S produces a pattern match \*/

      for (k=j; k<=match2[0]; ++k) printf(" (%d,%d)", match1[i], match2[k]);

   }

   printf("\n");

   /\* Clean up locally allocated memory \*/

   free(S); free(T1); free(T2); free(match1); free(match2);

}

int main ( int argc, char \*argv[] )

{

   int n, m;

   srand((unsigned int)time(NULL));

   if (argc >= 3) {

      n = atoi(argv[1]); m = atoi(argv[2]);

   } else {

      printf("n = |S| = "); scanf("%d", &n);

      printf("m = |T| = "); scanf("%d", &m);

   }

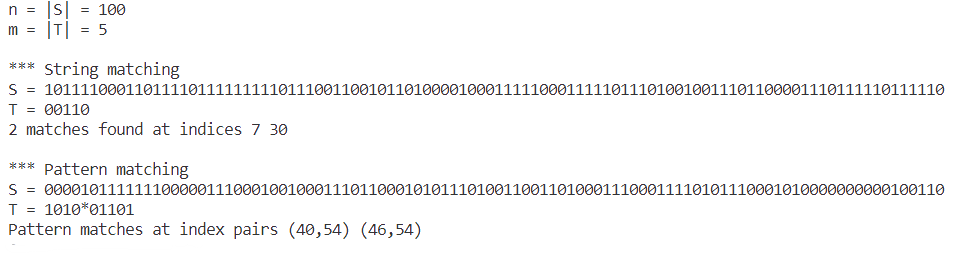
   printf("\n\*\*\* String matching\n"); strmatch(n,m);

   printf("\n\*\*\* Pattern matching\n"); patmatch(n,m);

   exit(0);

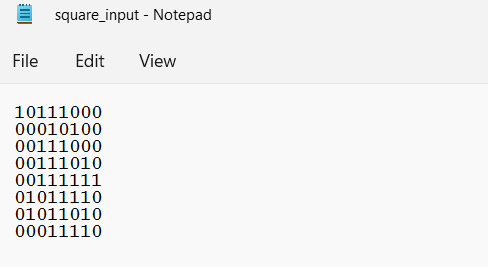
}

### Sample output



**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)

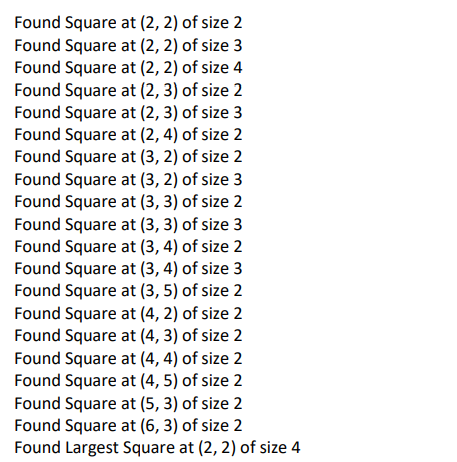
                if rows == cols:

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

                    largest = matcher.display\_position(positions)

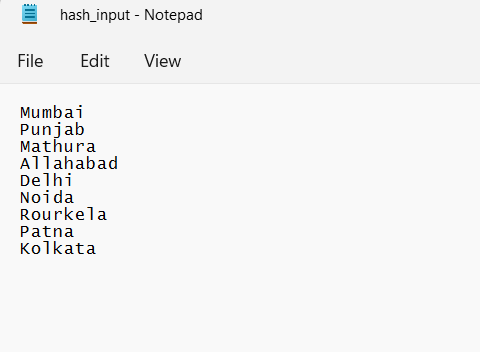
                    matcher.show\_largest(positions, largest)

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)

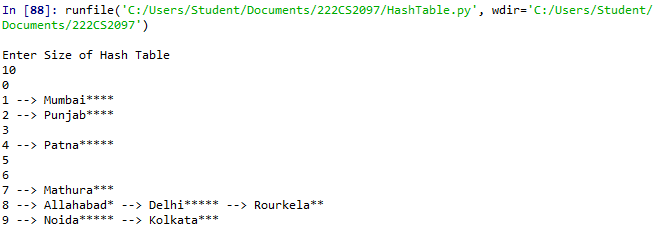
for token in modified:

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

    hash.insert(ascii\_sum, token)

hash.display\_hash()

Output:



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