**Bioinformatics1: Finding hidden messages in the DNA**

1. implement **PatternCount** (reproduced below).

     **Input:** Strings *Text* and *Pattern*.  
     **Output:** *Count*(*Text*, *Pattern*)

Algorith:.

**PatternCount**(*Text*, *Pattern*)

*count* ← 0

**for** i ← 0 to |Text| − |Pattern|

**if** *Text*(*i*, |*Pattern*|) = Pattern

*count* ← *count* + 1

**return** *count*

def count(strn, substrn):  
 sum =0  
 for i in range (len(strn) - len(substrn)+1):  
 for j in range(len(substrn)):  
 print("strn = ",strn[i+j], "substr = ", substrn[j])  
 input()  
 if strn[i+j] != substrn[j]:  
 print("no match")  
 input()  
 break  
 else:  
 if j != len(substrn)-1:  
 continue  
  
 print("match")  
 sum +=1  
 print("sum =", sum)  
 input()  
 print(sum)  
  
  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 s, sub = input("Please enter string then substring respectively: ").split()  
 count(s, sub)

#cleaned:

def count(strn, substrn):  
 total = 0  
 for i in range(len(strn) - len(substrn) + 1):  
 for j in range(len(substrn)):  
 if strn[i+j] != substrn[j]:  
 break  
 else:  
 if j != len(substrn)-1:  
 continue  
 total += 1  
  
 return total  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 s, sub = input("Please enter string then substring respectively: ").split()  
 print(count(s, sub))

2. **Code Challenge:** Solve the Frequent Words Problem.

* **Input:** A string Text and an integer k.
* **Output:**All most frequent k-mers in Text.

[Debug Datasets](http://bioinformaticsalgorithms.com/data/debugdatasets/replication/FrequentWords.zip)

**Note:**Please make sure that you return your answer as a collection of strings separated by spaces on a single line, as indicated by the sample output. In general, you should follow the formatting implied by the sample output in each problem.

**Sample Input:**

ACGTTGCATGTCGCATGATGCATGAGAGCT

4

**Sample Output:**

CATG GCAT

def frequency(strn, km):  
 knuc = []  
 for i in range(len(strn) - km +1):  
 nuc = ''  
 for j in range (km):  
 nuc += strn[j+i]  
 ''' print ("nuc = ", nuc)  
 input()  
 '''  
 knuc.append(nuc)  
 ''' print("list = ",knuc)  
 input()  
 '''  
 return max(set(knuc), key = knuc.count)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 s = file.read()  
 k = int(input("please enter lenght of k: "))  
 print(frequency(s, k))

#cleaned

def frequency(strn, km):  
 knuc = []  
 for i in range(len(strn) - km + 1):  
 nuc = ''  
 for j in range(km):  
 nuc += strn[j+i]  
 knuc.append(nuc)  
 return max(set(knuc), key=knuc.count)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 s = file.read()  
 k = int(input("please enter length of k: "))  
 print(frequency(s, k))

**Better answer**

from collections import Counter  
  
def frequency(strn, km):  
 knuc = []  
 for i in range(len(strn) - km + 1):  
 nuc = ''  
 for j in range(km):  
 nuc += strn[j+i]  
 knuc.append(nuc)  
 counter = Counter(knuc)  
 return counter.most\_common()  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 s = file.read()  
 k = int(input("please enter length of k: "))  
 print(frequency(s, k))

**Even better (final ver in n^2) :**

from collections import Counter  
  
  
def frequency(strn, km):  
 knuc = []  
 for i in range(len(strn) - km + 1):  
 nuc = ''  
 for j in range(km):  
 nuc += strn[j+i]  
 knuc.append(nuc)  
 counter = Counter(knuc)  
 maximum = counter.most\_common(1)[0][1]  
 for i in range(len(counter.most\_common())):  
 if counter.most\_common()[i][1] == maximum:  
 print(counter.most\_common()[i][0])  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 s = file.read()  
 k = int(input("please enter length of k: "))  
 frequency(s, k)

**A LOT better answer (actually the best one)**

from collections import Counter  
  
  
def frequency(strn, km):  
 knuc = []  
 for i in range(len(strn) - km + 1):  
 knuc.append(strn[i: i+km])  
 counter = Counter(knuc)  
 for i in range(len(counter.most\_common())):  
 if counter.most\_common()[i][1] == counter.most\_common(1)[0][1]:  
 print(counter.most\_common()[i][0])  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 s = file.read()  
 k = int(input("please enter k value: "))  
 frequency(s, k)

You can also use a dict to keep values(called frequency map)

def frequent(g, k):  
 freq\_map = {}  
 for i in range(len(g)):  
 seg = g[i: i + k]  
 if seg not in freq\_map:  
 freq\_map[seg] = 1  
 else:  
 freq\_map[seg] += 1  
  
 print(\*[key for key, val in freq\_map.items() if val == max(freq\_map.values())])  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("test", "r") as file:  
 gene = file.read()  
 km = int(input("please enter k:"))  
 frequent(gene, km)

**To be explored later(important)!!!**

def freq(strn, substrn):  
 i =0  
 count = 0  
  
 while i < len(strn):  
 j = strn.find(substrn, i)  
 print("i = ",i ,"j =", j)  
 input()  
  
 if j != -1:  
 i = j + 1  
 print("i = ",i ,"j =", j)  
 input()  
 count += 1  
 print("count increased, count =", count)  
 input()  
 else:  
 print(" supposedly j = -1")  
 return print("final count =",count)  
 return print("final count =",count)  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 s = file.read()  
 sub = input('enter sub: ')  
 freq(s, sub)

**3.The reverse complement of a string**

**4.  Solve the Pattern Matching Problem**.

* **Input**: Two strings, *Pattern* and *Genome*.
* **Output**: A collection of space-separated integers specifying all starting positions where *Pattern* appears as a substring of *Genome*.

[Debug Datasets](http://bioinformaticsalgorithms.com/data/debugdatasets/replication/PatternMatching.zip)

**Sample Input:**

ATAT

GATATATGCATATACTT

**Sample Output:**

1 3 9

Answer:

def positions(genome, substrn):  
 for i in range(len(genome) - len(substrn) + 1):  
 for j in range(len(substrn)):  
 if genome[j + i] != substrn[j]:  
 break  
 if j == len(substrn)-1:  
 print(i, end=" ")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 g = file.read()  
 sub = input("please enter pattern: ")  
 positions(g, sub)

**Code Challenge:** Solve the Clump Finding Problem (restated below). You will need to make sure that your algorithm is efficient enough to handle a large dataset.

**Clump Finding Problem:** Find patterns forming clumps in a string.

* **Input**: A string Genome, and integers k, L, and t.
* **Output**: All distinct k-mers forming (L, t)-clumps in Genome.

[Debug Datasets](http://bioinformaticsalgorithms.com/data/debugdatasets/replication/ClumpFinding.zip)

**Sample Input:**

CGGACTCGACAGATGTGAAGAACGACAATGTGAAGACTCGACACGACAGAGTGAAGAGAAGAGGAAACATTGTAA

5 50 4

**Sample Output:**

CGACA GAAGA

from collections import defaultdict  
  
  
def search(inseq, k, L, t):  
 lookup = defaultdict(list)  
 result = set()  
  
  
 for cursor in range(len(inseq) - k + 1):  
 seg = inseq[cursor:cursor + k]  
 # Testing Seg Val  
 print(f"seg ={seg}")  
  
 input()  
 # --------  
  
 # remove prior positions of the same segment  
 # if they are more than L distance far  
 while lookup[seg] and cursor + k - lookup[seg][0] > L:  
 print(f"lookup[{seg}] = {lookup[seg]},lookup[{seg}] and cursor + k - lookup[{seg}][0] = {lookup[seg] and cursor + k - lookup[seg][0] }")  
 input()  
 lookup[seg].pop(0)  
 print(f" condition applied lookup[{seg}] = {lookup[seg]}")  
 input()  
 lookup[seg].append(cursor)  
 print(f"condition not applied lookup[{seg}] = {lookup[seg]}")  
 input()  
 if len(lookup[seg]) == t:  
 result.add(seg)  
 print("result",result)  
 input()  
  
 print(result)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 g = file.read()  
 K, L, T = map(int, input("please enter k, l and t respectively:").split())  
 search(g, K, L, T)

**Week 2**

**Minimum Skew Problem:** *Find a position in a genome where the skew diagram attains a minimum.*

* **Input**: A DNA string *Genome*.
* **Output**: All integer(s) *i* minimizing *Skewi* (*Genome*) among all values of *i* (from 0 to |*Genome*|).

**Code Challenge:** Solve the Minimum Skew Problem.

[Debug Datasets](http://bioinformaticsalgorithms.com/data/debugdatasets/replication/MinimumSkew.zip)

**Sample Input:**

TAAAGACTGCCGAGAGGCCAACACGAGTGCTAGAACGAGGGGCGTAAACGCGGGTCCGAT

**Sample Output:**

11 24

To clarify and hopefully prevent some 0-based/1-based indexing confusion ...

If Genome has length n, then the skew array of Genome will have n+1 total values because the first element is always zero. This means that Skewi (Genome) corresponds to the number of G's minus the number of C's in the first i symbols of Genome.

But we tend to use 0-based indexing, so that the first i symbols of Genome range between positions 0 and i - 1.

def skew(strn):  
 i = 0  
 i\_val\_lst = []  
 for n in strn:  
 if n == "G":  
 i += 1  
 '''print("G found, i increased. i =", i)  
 input()'''  
 elif n == "C":  
 i -= 1  
 '''print(f"C found, i decreased, i = {i}")  
 input()'''  
 '''else:  
 print(f"i remains the same. i = {i}")  
 input()'''  
 i\_val\_lst.append(i)  
 '''print(f"list = {i\_val\_lst}")  
 input()'''  
 print(f"minimum = {min(i\_val\_lst)}")  
 input()  
  
 for j in range(len(i\_val\_lst)):  
 if i\_val\_lst[j] == min(i\_val\_lst):  
 print(f"answer = {j + 1}")  
 input()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 g = file.read()  
 skew(g)

#cleaned

def skew(strn):  
 i = 0  
 i\_val\_lst = []  
 for n in strn:  
 if n == "G":  
 i += 1  
 elif n == "C":  
 i -= 1  
 i\_val\_lst.append(i)  
  
 for j in range(len(i\_val\_lst)):  
 if i\_val\_lst[j] == min(i\_val\_lst):  
 print(j + 1, end=" ")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open(r"ros\_file", "r") as file:  
 g = file.read()  
 skew(g)

#to be explored later

def find\_minimum\_skew(genome):  
 skew\_values = [0]  
  
 min\_skew = 0  
  
 min\_positions = []  
  
 for i, nucleotide in enumerate(genome, start=1):  
  
 if nucleotide == 'G':  
  
 skew\_values.append(skew\_values[-1] + 1)  
  
 elif nucleotide == 'C':  
  
 skew\_values.append(skew\_values[-1] - 1)  
  
 else:  
  
 skew\_values.append(skew\_values[-1])  
  
 if skew\_values[-1] < min\_skew:  
  
 min\_skew = skew\_values[-1]  
  
 min\_positions = [i]  
  
 elif skew\_values[-1] == min\_skew:  
  
 min\_positions.append(i)  
  
 return min\_positions  
  
  
with open(r"C:\Users\LENOVO\Downloads\dataset\_30277\_10.txt", "r") as file:  
 genome = file.read()  
minimum\_skew\_positions = find\_minimum\_skew(genome)  
print("Positions with minimum skew:", minimum\_skew\_positions)  
end

# the ripe result of my 7anyaka, a bit (a lot) messy in terms of logic and execution. Some #### how it is also the most efficient out of all the previously mentioned codes (only 0.07 seconds to execute. much wow)

def skew(strn):  
 i = 0  
 minimum = 0  
 count = 1  
 lst\_min = []  
 for n in strn:  
 if n == "G":  
 i += 1  
 elif n == "C":  
 i -= 1  
 if i < minimum:  
 minimum = i  
 lst\_min = []  
 lst\_min.append(count)  
 count += 1  
 continue  
 if i == minimum:  
 lst\_min.append(count)  
  
 count += 1  
 return lst\_min  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open(r"C:\Users\LENOVO\Downloads\dataset\_30277\_10 (2).txt", "r") as file:  
 g = file.read()  
 print(\*skew(g), sep=' ')

**2.Solve the Approximate Pattern Matching Problem:**

We say that a *k*-mer *Pattern* appears as a substring of *Text* with at most *d* mismatches if there is some *k*-mer substring *Pattern'* of *Text* having *d* or fewer mismatches with *Pattern*, i.e., *HammingDistance*(*Pattern*, *Pattern'*) ≤ *d*. Our observation that a *DnaA* box may appear with slight variations leads to the following generalization of the Pattern Matching Problem.

**Approximate Pattern Matching Problem**: *Find all approximate occurrences of a pattern in a string.*

* ***Input****: Strings Pattern and Text along with an integer d.*
* ***Output****: All starting positions where Pattern appears as a substring of Text with at most d mismatches.*

**Code Challenge:** Solve the Approximate Pattern Matching Problem.

[Debug Datasets](http://bioinformaticsalgorithms.com/data/debugdatasets/replication/ApproximatePatternMatching.zip)

**Sample Input:**

ATTCTGGA

CGCCCGAATCCAGAACGCATTCCCATATTTCGGGACCACTGGCCTCCACGGTACGGACGTCAATCAAAT

3

**Sample Output:**

6 7 26 27

Answer:

def positions(genome, substrn, d):  
 for i in range(len(genome) - len(substrn) + 1):  
 hd = 0  
 for j in range(len(substrn)):  
 if genome[j + i] != substrn[j]:  
 hd += 1  
 if hd > d:  
 break  
 if j == len(substrn)-1:  
 print(i, end=" ")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 g = file.read()  
 sub = input("please enter pattern: ")  
 num = int(input("enter d: "))  
 positions(g, sub, num)

**Code Challenge:** Implement **ApproximatePatternCount**.

* **Input:**Strings *Pattern* and *Text* as well as an integer *d*.
* **Output:** *Countd*(*Text*, *Pattern*).

[Debug Datasets](http://bioinformaticsalgorithms.com/data/debugdatasets/replication/ApproximatePatternCount.zip)

**Hint:**Don't make this problem too hard if you solved the Approximate Pattern Matching Problem.

**Sample Input:**

GAGG

TTTAGAGCCTTCAGAGG

2

**Sample Output:**

4

def positions(genome, substrn, d):  
 count = 0  
 for i in range(len(genome) - len(substrn) + 1):  
 hd = 0  
 for j in range(len(substrn)):  
 if genome[j + i] != substrn[j]:  
 hd += 1  
 if hd > d:  
 break  
 if j == len(substrn)-1:  
 count += 1  
 return count  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 g = file.read()  
 sub = input("please enter pattern: ")  
 num = int(input("enter d: "))  
 print(positions(g, sub, num))

Given strings *Text* and *Pattern* as well as an integer *d*, we define *Countd*(*Text*, *Pattern*) as the total number of occurrences of *Pattern* in *Text* with at most *d* mismatches. For example, *Count*1(**AACAA**GCTG**ATAAACA**TTT**AAAGA**G, **AAAAA**) = 4 because **AAAAA** appears four times in this string with at most one mismatch: **AACAA**, **ATAAA**, **AAACA**, and **AAAGA**. Note that two of these occurrences overlap.

A **most frequent *k*-mer with up to *d* mismatches** in *Text* is simply a string *Pattern* maximizing *Countd*(*Text*, *Pattern*) among all *k*-mers. Note that *Pattern* does not need to actually appear as a substring of *Text*; for example, AAAAA is the most frequent 5-mer with 1 mismatch in AACAAGCTGATAAACATTTAAAGAG, even though AAAAA does not appear exactly in this string. Keep this in mind while solving the following problem.

**Frequent Words with Mismatches Problem**

*Find the most frequent k-mers with mismatches in a string.*

**Given:** A string *Text* as well as integers *k* and *d*.

**Return:** All most frequent *k*-mers with up to *d* mismatches in *Text*.

**Sample Dataset**

**ACGTTGCATGTCGCATGATGCATGAGAGCT**

**4 1**

**Sample Output**

**GATG ATGC ATGT**

def hamming\_dist(p, q):  
 if len(p) != len(q):  
 raise ValueError("strings must be of the same length")  
 else:  
 return sum( ch1 != ch2 for ch1, ch2 in zip(p, q))  
  
def mismatch\_frequency(g, k, d):  
 freq\_map = {}  
 seg\_list = []  
 for i in range(len(g) - k + 1):  
 seg = g[i: i + k]  
 if seg not in freq\_map:  
 seg\_list.append(seg)  
  
 for k1 in set(seg\_list):  
 for k2 in seg\_list:  
 # showing k1 and k2 vals:  
 print(f"key 1 = {k1}, key 2 = {k2}")  
 input()  
 # ------------------------------------  
 if hamming\_dist(k1, k2) <= d:  
 #testing hamming\_dist  
 print(f"hamming distance = {hamming\_dist(k1, k2)}")  
 # -------------------------------------------------  
 if k1 not in freq\_map:  
 freq\_map[k1] = 1  
 else:  
 freq\_map[k1] += 1  
 #freq\_map[k2] += 1  
 # testing correct increment of values  
 print(f"freq\_map[{k1}] = {freq\_map[k1]}")  
 #print(f"freq\_map[{k2}] = {freq\_map[k2]}")  
 input()  
 # -----------------------------------------  
  
 print("freq\_map", freq\_map)  
 input()  
 print("results:")  
 print([key for key,val in freq\_map.items() if val == max(freq\_map.values())])  
  
  
  
  
  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("test", "r") as file:  
 gene = file.read()  
 K, D = map(int, input("please enter k and d respectively:").split())  
 mismatch\_frequency(gene, K, D)

#cleaned

def hamming\_dist(p, q):  
 if len(p) != len(q):  
 raise ValueError("strings must be of the same length")  
 else:  
 return sum(ch1 != ch2 for ch1, ch2 in zip(p, q))  
  
  
def mismatch\_frequency(g, k, d):  
 freq\_map = {}  
 seg\_list = []  
 for i in range(len(g) - k + 1):  
 seg = g[i: i + k]  
 if seg not in freq\_map:  
 seg\_list.append(seg)  
  
 for k1 in set(seg\_list):  
 for k2 in seg\_list:  
 if hamming\_dist(k1, k2) <= d:  
 if k1 not in freq\_map:  
 freq\_map[k1] = 1  
 else:  
 freq\_map[k1] += 1  
 print(\*[key for key, val in freq\_map.items() if val == max(freq\_map.values())])  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("test", "r") as file:  
 gene = file.read()  
 K, D = map(int, input("please enter k and d respectively:").split())  
 mismatch\_frequency(gene, K, D)

#trial

def ImmediateNeighbors(p):  
 neighborhood = set()  
 for i in range(len(p)):  
 symbol = p[i]  
 for nucleotide in p:  
 if nucleotide != symbol:  
 print(f"nuc ={nucleotide}")  
 input()  
 p = list(p)  
 p[i] = nucleotide  
 p = ''.join(p)  
 neighbor = p  
 print(f"neighbour = {neighbor}")  
 input()  
 neighborhood.add(neighbor)  
 print(neighborhood)

def HammingDistance(p,q):  
 if len(p) != len(q):  
 raise ValueError("strings must be of the same length")  
 else:  
 return sum(ch1 != ch2 for ch1,ch2 in zip(p,q))  
  
  
  
def Neighbors(Pattern, d):  
 if d == 0:  
 return Pattern  
 if len(Pattern) == 1 :  
 return "A", "C", "G", "T"  
 Neighborhood = set()  
 Pattern = Pattern[1:]  
 SuffixNeighbors = Neighbors(Pattern, d)  
 for Text in SuffixNeighbors:  
 if HammingDistance(Pattern[-1], Text) < d:  
 for nucleotide in Text:  
 Neighborhood.add(nucleotide + Text)  
 print(nucleotide, Text, Neighborhood)  
 input()  
 else:  
 Neighborhood.add(Pattern[0] + Text)  
 print(Neighborhood)  
 input()  
 return Neighborhood  
print(Neighbors("CAA",1))

**Neighborhood code 1:**

def neighborhood(p):  
 neighbors = set()  
 nucs = ["A", "C", "G", "T"]  
 for i in range(len(p)):  
 for nuc in nucs:  
 if p[i] == nuc:  
 neighbors.add(p)  
 else:  
 q = list(p)  
 q[i] = nuc  
 q = ''.join(q)  
 neighbors.add(q)  
 print(\*neighbors)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 strn = input("please enter the needed string:")  
 neighborhood(strn)

**Neighborhood code 2(the solution):**

def HammingDistance(p, q):  
 if len(p) != len(q):  
 raise ValueError("strings are not of the same length")  
 else:  
 return sum(ch1 != ch2 for ch1,ch2 in zip(p,q))  
  
def Neighbors(Pattern, d):  
 nucs = ["A", "C", "G", "T"]  
 if d == 0:  
 return Pattern  
 if len(Pattern) == 1:  
 return nucs  
 Neighborhood = set()  
 print("Pattern: ", Pattern)  
 SuffixNeighbors = Neighbors(Suffix(Pattern), d)  
 print("Now we go. Pattern:", Pattern)  
 #print(SuffixNeighbors)  
 for Text in SuffixNeighbors:  
 if HammingDistance(Suffix(Pattern), Text) < d:  
 for nuc in nucs:  
 Neighborhood.add(nuc + Text)  
 else:  
 Neighborhood.add(Pattern[0] + Text)  
 return Neighborhood  
  
  
def Suffix(Pattern):  
 S=list(Pattern)  
 S.pop(0)  
 S= ''.join(S)  
 return S  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 p = input("please enter pattern:")  
 d = int(input("please enter d:"))  
 print(Neighbors(p, d))

**Week 3**

To get profile matrix of a number of dna sequences

def profile(r):  
 profile\_matrix = {"A": [], "G": [], "C": [], "T": []}  
 for i in range(len(r[0])):  
 for element in profile\_matrix:  
 profile\_matrix[element].append(0)  
 for j in range(len(r)):  
 profile\_matrix[r[j][i]][i] = (profile\_matrix[r[j][i]][i] + 1)  
 for key in profile\_matrix.keys():  
 print(key, ":", [z / len(r) for z in profile\_matrix[key]])  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open("ros\_file", "r") as file:  
 rows =file.read().splitlines()  
 profile(rows)

Neighbourhood problem:

def HammingDistance(p, q):  
 if len(p) != len(q):  
 raise ValueError("strings are not of the same length")  
 else:  
 return sum(ch1 != ch2 for ch1,ch2 in zip(p,q))  
  
def Neighbors(Pattern, d):  
  
 nucs = ["A", "C", "G", "T"]  
 if d == 0:  
 return Pattern  
 if len(Pattern) == 1:  
 return nucs  
 Neighborhood = set()  
 SuffixNeighbors = Neighbors(Suffix(Pattern), d)  
 for Text in SuffixNeighbors:  
 if HammingDistance(Suffix(Pattern), Text) < d:  
 for nuc in nucs:  
 Neighborhood.add(nuc + Text)  
  
 else:  
 Neighborhood.add(Pattern[0] + Text)  
  
 return Neighborhood  
  
  
def Suffix(Pattern):  
 S = list(Pattern)  
 S.pop(0)  
 S = ''.join(S)  
 return S  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 p = input("please enter pattern:")  
 d = int(input("please enter d:"))  
 print(\*Neighbors(p, d))