```c


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <ctype.h>
#include "stringBuilder.h"


#define ALPHABET_LENGTH 26
double A[ALPHABET_LENGTH]; // letter frequencies
int ciphertext_length;


int Mod(int a, int b)
{
    return (a % b + b) % b;
}

int GCD(int x, int y)
{
    if (y == 0) return x;
    return GCD(y, x % y);
}

char* Cipher(char* input, char* key, bool encipher)
{
    int keyLen = strlen(key);

    for (int i = 0; i < keyLen; ++i)
        if (!isalpha(key[i]))
            return ""; // Error

    int inputLen = strlen(input);
    char* output = (char*)malloc(inputLen + 1);
    int nonAlphaCharCount = 0;

    for (int i = 0; i < inputLen; ++i)
    {
        if (isalpha(input[i]))
        {
            bool cIsUpper = isupper(input[i]);
            char offset = cIsUpper ? 'A' : 'a';
            int keyIndex = (i - nonAlphaCharCount) % keyLen;
            int k = (cIsUpper ? toupper(key[keyIndex]) : tolower(key[keyIndex]))
              - offset;
            k = encipher ? k : -k;
            char ch = (char)((Mod(((input[i] + k) - offset), 26)) + offset);
            output[i] = ch;
        }
```

```
52          else
53          {
54              output[i] = input[i];
55              ++nonAlphaCharCount;
56          }
57      }
58
59      output[inputLen] = '\0';
60      return output;
61  }
62
63  char* Encipher(char* input, char* key)
64  {
65      return Cipher(input, key, true);
66  }
67
68  char* Decipher(char* input, char* key)
69  {
70      return Cipher(input, key, false);
71  }
72
73
74
75
76
77
78  int GetKeyLength(int ioc[2000])
79  {
80      int max1 = 0;
81      int max2 = 0;
82      int index1 = 0;
83      int index2 = 0;
84
85      for (int i = 0; i < 2000; i++)
86      {
87          if (ioc[i] > max1 && ioc[i] > max2)
88          {
89              max2   = max1;
90              index2 = index1;
91              max1   = ioc[i];
92              index1 = i;
93          }
94      }
95
96      return GCD(index1, index2);
97  }
98
99
100 /*****************************************************************************
101 FUNCTION        : encrypt
102
103 DESCRIPTION     : This function encrypts a character based on a key as the
```

```
104 parameter.
105
106 INPUT           : Type          : char
107 : Description   : The character to encrypt.
108
109 : Type          : int
110 : Description   : The shift key.
111
112 OUTPUT          : Type          : char
113 : Description   : The key-shifted chracter
114 ********************************************************************************/
115 char encrypt(char ch, int key)
116 {
117     if (!isalpha(ch)) return ch;
118     char offset = isupper(ch) ? 'A' : 'a';
119     return (char)((((ch + key) - offset) % 26) + offset); // shift cipher
120 }
121
122
123 /********************************************************************************
124 FUNCTION        : decrypt
125
126 DESCRIPTION     : This function decrypts each character of the ciphertext using the
127                   given key parameter.
128
129 INPUT           : Type          : char *
130                 : Description   : The ciphertext.
131
132                 : Type          : int
133                 : Description   : The shift key.
134
135 OUTPUT          : Type          : char *
136                 : Description   : The decrypted text
137 ********************************************************************************/
138 char * decrypt(char * text, int key)
139 {
140     int text_length = strlen(text);
141     char * plaintext = (char*)malloc(text_length + 1);
142
143     for (int i = 0; i < text_length; i++)
144         plaintext[i] = encrypt(text[i], key);
145
146     plaintext[text_length] = '\0'; // add null termination character
147
148     return plaintext;
149 }
150
151
152
153
154 /********************************************************************************
```

```
155 FUNCTION        : innerProduct
156
157 DESCRIPTION     : This function uses the frequencies of letters expected
158 in an english message that has been Caeser-shifted i
159 letters to the left by a 26-dimensional vector, A.
160
161 One of these vectors should agree fairly closely with the
162 frequencies of letters we see in our ciphertext.
163 Which vector that is tells us the shift amount for our sampling,
164 and the first letter of our keyword.
165
166 To find the vector in the previous list above that most closely
167 matches the vector u, we recall that the dot product of two
168 vectors is connected to the angle θ between those two vectors
169 in the following way:
170
171 W.A=|W||A|cosθ
172
173 If we want to find the two vectors W and Ai that most closely
174 match, we want to find the two vectors with the smallest
175 angle between them.
176
177 Noting that smaller angles produce larger cosine values and
178 also noting that the magnitude of the denominator is the same
179 for every vi as the same 26 numbers are involved each time
180 (just in different orders), we can simply seek the two vectors W
181 and Ai whose dot product is largest.
182
183
184 INPUT           : Type          : int[]
185 : Description   : The letter frequencies of the ciphertext.
186
187 : Type          : int *
188 : Description   : A reference to the encryption key.
189
190 OUTPUT          : Type          : double *
191 : Description   : The array of innerproducts.
192 ********************************************************************************/
193 double * innerProduct(int W[], int * key)
194 {
195     double inner_product[ALPHABET_LENGTH] = { 0 };
196     double sum = 0;
197     int j;
198
199
200     for (int i = 0; i < ALPHABET_LENGTH; i++)
201     {
202         for (j = 0; j < ALPHABET_LENGTH; j++)
203             sum += W[j] * A[(j + i) % ALPHABET_LENGTH]; // shift the frequency
                  array
204
205         inner_product[i] = sum / ciphertext_length;
```

```
206
207         // find the largest innerproduct. This will be the key.
208         if (inner_product[*key] < inner_product[i]) *key = i;
209
210         // reset counter and sum
211         j = 0;
212         sum = 0;
213     }
214
215     printf("Inner products: \n\n");
216     for (size_t i = 0; i < 26; i++)
217         printf("W[%d]\t=  %f\n", i + 1, inner_product[i]);
218
219     return inner_product;
220 }
221
222
223
224 int main()
225 {
226
227 #pragma region Completed
228     // english alphabet letter frequncies
229     A[0] = 0.08167;
230     A[1] = 0.01492;
231     A[2] = 0.02782;
232     A[3] = 0.04253;
233     A[4] = 0.12702;
234     A[5] = 0.02228;
235     A[6] = 0.02015;
236     A[7] = 0.06094;
237     A[8] = 0.06996;
238     A[9] = 0.00153;
239     A[10] = 0.00772;
240     A[11] = 0.04025;
241     A[12] = 0.02406;
242     A[13] = 0.06749;
243     A[14] = 0.07507;
244     A[15] = 0.01929;
245     A[16] = 0.00095;
246     A[17] = 0.05987;
247     A[18] = 0.06327;
248     A[19] = 0.09056;
249     A[20] = 0.02758;
250     A[21] = 0.00978;
251     A[22] = 0.02360;
252     A[23] = 0.00150;
253     A[24] = 0.01974;
254     A[25] = 0.00074;
255
256     char * ct =                                                    ↵
        "YFLWIMJRGYEXMAFQEEMQIMTRMFGYXPRRUCDJSEFCFQCSHPEXWVFFFSJZZPKGEIDLPMWUGCDXEY ↵
```

```
        KQXIRQDPRSVZNSFPMADBMKEVMQFXLRAPUKLGKGSLXGGCHMZVCQQRWRNDEPYZACDMRTEGDIXUZRE
        IIZDBFSJVMBQBTEDQEMSANLXCMASFAWIJNLPIVSTJFEAAXCKIWBEFQVWEDKMVONAJQICRRBUJJR
        QCZXJENKMRCBSFQVABLYZWXUZRULEIDCHIVXMMIRXUDGZXIARCBSARQMRWXVKJZIWFRFQTSFRCE
        WIQVFUGLADTQVXUDJQWWPNLHICRCRTIMZOPQWWVNLAJEJHJPYRGZKQHWCHPUXMAZLQBUHHQUXIY
        XAUZMYHQQHFBCWMPPGGCEIXUHLSWEEDZGVRGHLFSQLLCYSVLHQTEPYMCHIVSNPSIXGGCYWLRFPQ
        IXRCKQAMGGYRIAJNPPWSSOJQEWNMRIIPPNKQMRNKMIGPRZPHSMPDYZHMFZLWMRGNYNEWXDROLEV
        QDQIPVMEPMWGHLOXPLFJMHXUZRULEQZAOITGDBVSLARGZZMGZRUSRZQQOEZRMBUWLTZTQQIFNKQ
        XINZLPLIEECIUYVDRDIQNQIELIVFFFIRRCKKJMERRUQTEDQEMSANDTIVNRYFLSENSSLPLEYEGMA
        ZRURKJNKMREAZNBVIPHYFMZRKGEXIADPUWEYVYKWWGHKGPEGHLSERQHBQWGEHZQHMAZFGQSENSE
        QEAMCDGIESYURMABGPIRGRMRQCPNLHEPRRAQRXUNKQMRNVYKALVBFUJPNSRQVQLRCXJKEDYFPCN
        LSEIHZXFAWXRRQVSLANDOSYERCSSSQECXPSJSFAYKUGCUWGBTJPLEECJKFIPZJXIHNAPUPPVZLF
        GSAUCDWEGHMZEPVRRMXXUZRYSQRMRMAIYKPQQIZACDIHINGOIJYNYFIHGGPAYKUSFQSTRMDDIRP
        GUURHBVLQEVNSFMRHGGCZCSHKJIVMGDRAXLROPURGRRQMJXRQRQEEYEPQHMYKUDMXRSMXEHLSYP
        QMARRQVJBQRTIWRBMZHHNXKKWIYEMDWLNKJIIANHRGRXVKUQLINQDDSQGGCBVMABCEWMABYEISS
        ZPQJYFZJXEHLSYPQMARRQVQVFFFSTRMGFXLREGDWXQZWMRHZQQOVSFAGQXLRRCOSRQSFQRXUDPQ
        WXUDBGGLRRQMFSHSRTIWPGMAPJRSCFLIEDUMWXUDKGVQHQMREQNMQHSMPDYZHXUDLYVWVMEXIXU
        NPBWVBRCURVROJKCIFBCDXEVMJKEJGDPFIEJHJXHSDTGFIARKJKSYNQCESXUNSSLXSTJMPJEDBP
        IEESFQJVRMATAMACMIWAHMEATIAZJUXXYDUUHIEZLPELNMBESQRVFUXIUZGDIHBKBXEHLVGFLEF
        NKQALNSKMWXRQDGPGNRRAJJRZRGVIFRRQTTRCMGXSSHRARXBSFQPEJMYYERSNJXSARCFQVEFTES
        IWGHMZSJQDDQVIABCURLVRKMRRRQKDWMAFJQXLBQNSVIRSCPQIJHRTIJSTQUSRJGWUJMGHQZXXB
        NBQPMTGRRYPGNQQICBTYSEMALPTEWGHLSWESSCDEPYSFQWILDYDWEYEPQHHNQJURKZQFMWXVMEE
        QCUTQNERQHJASORCUUXLFNKQGYEHMEMXLZRMPJEDBPEVYHLSLIPDPFEMAKWEXVHBIMVEGGCDEPV
        DLZSXRHBUHRBSUARHRQYFNSUMMNNIPSGZKXBGGEFINQBUXANRMZISSSFQPSAFCEXEACZXEGXDQF
        MLNUCQZIERCQRLRVMDIKBKBDMQZDBBMRPDLQDEACFMHEPTPUSYFHKBEWFHTUXCBEDQEXHQCUXWG
        QSOOQRSFMXLRLGSLXYNMWREGTPMPSAZQFEKRASFAEFRRDERTDJKSYGNDBPEPDGZVINKJUJIUHQH
        SMPDUMWVNSFQVHRDNMRHHMAFYSHRFQTPNBCPEABNBQRLNMBURQV";
257     //char * ct =
        "DWGFQRVPLNYIEQBYUOXFVHPGXFIGXNTKZSNRKKXSTUTLKLDAENUPDULVSIUMAVRGEFKYEQXCCV
        NOBZWJSOHTPVGKPEFRRKZFIBCOHCFRLRHPSZECDIUYZIAVIVSENGHTTIOHAXNGSEIGLHGBQDVSY
        QERIASTJSEINHKGNBOVGOVSZLVKENIOUEXOUSFYVCTJOIAFIERKBEEQAUUQTLDUTOOOTCIUOFHR
        WAPNIRVIIPQKEFXTKCYRFXNIVQTUDRRCNUGLHCDPORHIVWQAAEOKBATFWRWQSEQWIUCTOHADGBE
        IPPNPYFSNNBWDUTVHSWQSEFIIXOMWVADKNQASAAURQDNRRQCEMRLAUSFPBHSKLXEGWAVWDSVCGN
        OFHBGPUWUNQLAUNQRNCGGNTAQHHGCAMRUAPDMSGXCKNQABUDGWANVPCCVBOFHEUCUOAPNFSRTUP
        TYODEFDWCCUTADTCVEOCDSUSNLRIHCDEHRBIIRFHNKEVKWEAWETYINYXFGSIAFPBQEFTBTXRYGN
        QIHGCQTUTOTSQSGDPQSDOGLHGXTIFDWPGARQHDKCFRNRTGNYEPDMGRQSNXDPYITBTXCWUNRIHGM
        AFSTEEEBSZNDGKDPBXRQDIHNIOPOMRGWIUDTETDOFYRTUPTPYITUPTYOWNBLADYGTGWEEYOOBWL
        CVMTUPTOSEEEPBNOOOPDCTSQDCDITYFFYXPRKZTYNHGVMUTWEFGUTUPPRKDEAIEPTAYZTNVBMIF
        XNIRUSNGMUDAHRPVGXUNZDCMNQSCPITSZWUPTKMAUYSNQDNUGROPCUDRGTJOIOEHTRYESVQLGDM
        SGTAPNMNLLAASEAVSWKDTIARRGKEIAVCQVPNRHSCCYRFXNIVQTUDRRDAOXWETMAFSTEWZETNXRU
        GUTUWETSPOADTUOQWUPTAYGEKEEEDFOSXNFEZLRHSAYGCBCSKNQRVILKUQLLIHCDIEFWANVPIFR
        OXODACPCMOFOSHTTIOHAXNGYZTUTCQPRERIRCIBOVGOVGMSFDBGBQDNIOPMQCBBEEYYEZNFTSQN
        QWEUKUDFAIRZUNTWIUKDMFIHTYGGUBIPOZEIDUUPMCUTZRKEAYAOYWQTBXNVODEFIMACQLSXNOI
        OOSUEGMGPFPNFSIIYARGCBEPIYQEDCBROVRQRRXSKDMBNGGCSZHRLAUCAQHPIPDXYUJMQBAUFIH
        CDUWNHFQBOEQIONKGGUPNFGQWRCTVYSEGWETDATUTDTKIIAVRQYYWUTRGDTEPDFHOQCHESCXPTE
        PYTOYAVCEFEZDVHTWBNEQPSYOTAQAEHDFHRBPQSDOGBAFOYEETCCZUTHAAVOFHRHCGXQOSIHGXU
        GUIBGPARRAIUDQNVCGXODYPPRGPGLYNAPNHEEXFASZGGWERYEIGXOPYRTUTVCBUOHHCWZESBBRU
        MMVRCDKCTSGDOFLKTUTTTKKAASPQEDEQDUVIQSGWEPCTEPPMGKORBHSVYFHRLIPNAWJWETOKOHH
        AVGUTUBAFOYOVHENVQCLCTJSMYRHHGBQAETTJOFHETEEEBSNCDVRQCHEOPDTEZPNVOXPVTCGRML
        SSRWXWTUPTYYGLQQEOBXAJGEPMQCNKEPNUSUHAPNFHRDNGYZTUTTTKKJBWNEKHEASIUREIFPWJS
        YPHIIVNAWAIHGBQGBDDQXQTJDTJBQESDUTPUVRQUVGTEETTJOZIFIHGMGPBUMTSZGYTTJYDPUTD
        QOENBITCUQCBUFGOFHRCANVMRRPCEYGNGTDHYDOATMQWQNGBYHBUEASWKDTIAUIPSFEPPRGRQTB
        DKCNDOCDRVGAFEDMVRQGEDUPNEIATAEROUCHECVUNTIHGWGPVCSGZMRNIEVOETGJBGCFAFIIPQQ
```

```
        APWIPDGRAPSJOPIQHOJSEPUNSKYSNBBYWXPEELEPDMCHGIQEECUPNIOMNRMPTOESVDNIKFHRGEF
        DTEETTJKFIPPNQXXYQTSEBUBRPSJKXFCJZBVQDNCDJKXFETLKOH";

    int shift = 1;
    int indexPos = 0;
    ciphertext_length = strlen(ct);

    // Kerckhoff's Method Part A:
    int IOC[2000] = { 0 }; // Incedents of Coincidence

    for (int i = shift; i < ciphertext_length - 1; i++)
    {
        for (int j = i; j < ciphertext_length; j++)
        {
            if (ct[j] == ct[indexPos]) IOC[shift] += 1;
            indexPos++;
        }

        shift++;
        indexPos = 0;
    }


    printf("%s\n\n", ct);
    printf("Number of letters in the text : %d\n\n", ciphertext_length);
    printf("The number of coincidences N(n) as a function of the number of shifts
       n :\n\n");

    for (int i = 1; i < 21; i++)
        printf("N(%d) = %d\n", i, IOC[i]);

    int keyLength = GetKeyLength(IOC);
    printf("\n\nVery likely the key length is %d.\n", keyLength);

    // GOOD UP TO HERE! //

#pragma endregion


    // Kerckhoff's Method Part B:
    /*
        1. Assume the key length is l.
        2. Split the ciphertext into l parts.
        3. For i = 1 to l
            a) Treat part i as the ciphertext resulted form a shift cipher.
            b) Decrypt part i using the method for cracking a shift cipher.
            c) If i = l, break; Else i++ and go back to step 3.
    */

    int j;
    for (int i = 0; i < keyLength; i++)
    {
```

```
307         j = i;
308         stringbuilder* ciphertext = sb_new();
309         for (; j < ciphertext_length; j+= keyLength)
310             sb_append_ch(ciphertext, ct[j]);
311
312         sb_make_cstring(ciphertext);
313         //printf("%s\n", sb_make_cstring(ciphertext));
314
315         int ct_length = strlen(sb_make_cstring(ciphertext));
316         int W[ALPHABET_LENGTH] = { 0 };
317         int key = 0;
318
319         // count occurences
320         for (int i = 0; i < ct_length; i++)
321             W[(int)sb_make_cstring(ciphertext)[i] - 65] += 1;
322
323         // compute inner product W . Ai, and find the key
324         double * inner_product = innerProduct(W, &key);
325
326         // get each kth letter and form a new array....
327         char * pt = decrypt(sb_make_cstring(ciphertext), key);
328
329         printf("key %d : %d\n", i ,key);
330     }
331
332
333     //printf("A likely encryption key can be obtained as : key = %s", key);
334     //printf("The plaintext recovered with the above key is as follows : \n%s",
          pt);
335
336     getchar();
337
338     return 0;
339 }
```