1 Problem statement

ElGamal. Decrypt the ciphertext from the table 7.4 page 305 (6.3 page 278 in the third edition), which was obtained by an application of the ElGamal Cryptosystem 7.1 page 257 (6.1 page 235). The parameters of the system are $p = 31847 = 1 + 2 \cdot 15923$ (15923 is prime), $\alpha = 5$, $a = 7899$, and $\beta = 18074$. Each element of $\mathbb{Z}_p$ in the range (0,17575) represents three alphabetic characters as in Exercise 6.13 page 247 (5.12 page 227). You have to use square-and-multiply algorithm for modular exponentiation, and the Extended Euclid Algorithm or other not-by-force algorithm for calculating modular inverses. You may use parts of the code from previous course assignments.

Shanks. What are the secret values of parameter k used for encryption? Use both Shanks's algorithm and brute force (for verification) to find them. Note that k's are not needed for the decryption. In this toy example, they can be found with the help of any discrete logarithm algorithm. Find the first 30 values of k.

2 Brief explanation

2.1 ElGamal

Definition. Let p be a prime such that the DISCRETE LOGARITHM problem in $(\mathbb{Z}_p^*, *)$ is infeasible, and let $\alpha \in \mathbb{Z}_p^*$ be a primitive element. Define the set of all keys $\mathcal{K}$ as

$$\mathcal{K} = \{(p, \alpha, a, \beta) : \beta \equiv \alpha^a \bmod p\}. \tag{1}$$

The values (p, α, β) are the *public key*, and a is the *private key*.

Encryption. For $K = (p, \alpha, a, \beta)$, a plaintext message x, and a secret random number $k \in \mathbb{Z}_{p-1}$, we define the encryption function e_K as

$$e_K(x, k) = (y_1, y_2), \tag{2}$$

where

$$y_1 = \alpha^k \bmod p \text{ and} \tag{3}$$

$$y_2 = x\beta^k \bmod p. \tag{4}$$

Decryption. The public key is $(p, \alpha, \beta) = (31847, 5, 18074)$, and the private key is $a = 7899$. The ElGamal decryption function is given by

$$d_K(y_1, y_2) = y_2(y_1^a)^{-1} \bmod p, \tag{5}$$

and we substitute the given values into this:

$$d_K(y_1, y_2) = y_2(y_1^{7899})^{-1} \bmod 31847 \tag{6}$$

2.2 Shanks

We give Shanks's algorithm below. This solves for a in

$$\alpha^a = \beta \bmod p. \tag{7}$$

In this problem, we know $\alpha = 5$ and $p = 31847$. We also know $y_1 = \alpha^a \bmod p$. So given a known y_1, we solve for a in

$$y_1 = 5^a \bmod 31847. \tag{8}$$

Algorithm 1 Shanks's algorithm for (G, p, α, β). This operates on $\mathbb{Z}_p$.

1: $m \leftarrow \lceil \sqrt{p} \rceil$
2: For all $0 \leq j \leq m-1$, compute $\alpha^{mj} \bmod p$.
3: For all $0 \leq i \leq m-1$, compute $\beta \cdot (\alpha^{-1})^i \bmod p$.
4: Sort the m ordered pairs $(j, \alpha^{mj} \bmod p)$ and $(i, \beta \cdot (\alpha^{-1})^i \bmod p)$ w.r.t. the second coordinate to create lists L_1 and L_2.
5: Search for identical second coordinates in L_1 and L_2; i.e. find two pairs such that y's are equal. So find $(j, y) \in L_1$ and $(i, y) \in L_2$.
6: Compute $a = \log_\alpha \beta = (m \cdot j + i) \bmod (p-1)$.
7: **return** a

3 Original plaintext (with spaces and punctuation)

She stands up in the garden where she has been working and looks into the distance. She has sensed a change in the weather. There is another gust of wind. A buckle of noise in the air and the tall cypresses sway. She turns and moves uphill towards the house climbing over a low wall, feeling the first drops of rain on her bare arms she crosses the loggia and quickly enters the house.

4 List of recovered values of k

(The Shanks method and the brute-force method give the same values of k.)

k00	29705	k15	25197
k01	28841	k16	31568
k02	18076	k17	22194
k03	21011	k18	18381
k04	478	k19	21976
k05	1576	k20	3815
k06	20710	k21	23219
k07	29302	k22	22519
k08	29115	k23	11024
k09	29705	k24	5312
k10	2500	k25	17622
k11	7195	k26	16220
k12	11446	k27	15385
k13	23119	k28	25967
k14	17240	k29	478

5 Source code

Listing 1. crypto-hw01.c

```c
// Advanced Crypto HW01 // Hannah Miller // 2020-01-29
//
// To run
//     clang crypto-hw01.c -lm -o crypto-hw01.out
//    ./crypto-hw01.out

// Standard libraries
#include <limits.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


//////////////////////////////////////////////////////////////////
// Square-and-multiply algorithm for modular exponentiation.
// This computes r = (x^a) mod p.
//
// Possibly useful:
//     https://graphics.stanford.edu/~seander/bithacks.html
//
    http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html
//
int sqmult(int a, int p, int x) {
  if (a==0)  // check if a=0 (used in Shanks)
    return 1;  // n^0 = 1, so just return that

  int t = ceil(log2(a));  // max index of exponent in base 2
  int r = x;  // initialize; first bit is always 1

  for (int i=t-1; i>0; i--) {  // scan MSB-1 to LSB
    //printf("i, %d\n", i);
    r = (r*r) % p;  // always square r and take mod p
    if (a >> (i-1) & 1)  // if the exponent bit is 1...
      r = (r*x) % p;  // ...then also multiply by x and take mod p
  }

  return r;  // return the remainder
}


//////////////////////////////////////////////////////////////////
// Compute inverses using the EEA.
//
```

```c
// Input  : Positive integers r0 and r1 with r0 > r1.
// Output : Pointer to an array with both inverses.
//          Computes 1 = s*r0 + t*r1.
//
int * compute_inverses(int r0, int r1) {

  // Enforce the r0 > r1 condition
  if (r0 < r1) {
    int tmp = r0;
    r0 = r1;
    r1 = tmp;
  }
  //printf("    check: r0, %d, r1, %d\n\n", r0, r1);

  // Initialize
  int s[3];
  int t[3];
  int r[3];

  s[0] = 1;   s[1] = 0;
  t[0] = 0;   t[1] = 1;
  r[0] = r0;  r[1] = r1;  r[2] = r0 % r1;

  // Loop
  while (r[2] > 0) {
    // Compute
    r[2] = r[0] % r[1];
    int q = (r[0] - r[2]) / r[1];

    s[2] = s[0] - q * s[1];
    t[2] = t[0] - q * t[1];

    // Update
    s[0] = s[1];  s[1] = s[2];
    t[0] = t[1];  t[1] = t[2];
    r[0] = r[1];  r[1] = r[2];

    // Print
    //printf("r0, %+04d, s0, %+04d, t0, %+04d\n", r[0], s[0], t[0]);
  }

  // Check for negative inverses and update if required
  if (s[0] < 0)
    s[0] = s[0] + r1;  // s0 = r0^-1
  if (t[0] < 0)
    t[0] = t[0] + r0;  // t0 = r1^-1

  // Print the final answer
```

```
  //printf("\nout of loop: r0, %d, s0, %d, t0, %d\n\n", r[0], s[0], t[0]);

  // Declare the output array
  static int out[2];
  out[0] = s[0];  // r0 inverse
  out[1] = t[0];  // r1 inverse

  return out;
}


///////////////////////////////////////////////////////////////////
// Given an integer a, compute the 3-letter string from its "base 26"
// representation as shown in ./slides/02-ElGamal-Shanks-dl0.pdf
//
void num2str(int a) {
  int r;  // remainder in a = q*b + r  ==>  a = q*26 + r
  char s[3];  // the output string

  for (int i=3; i>0; i--) {  // count down to fill the array nicely
    r = a % 26;  // compute the remainder
    a = (a-r)/26;  // shift the string by dividing by 26
    //printf("i-1, %d, r, %02d, a, %02d\n", i-1, r, a);
    s[i-1] = r+65;  // put the character into the `s` array
  }

  printf("%s\n", s);  // print the result

  /* for (int i=0; i<3; i++) */
  /*   printf("%s", *s[i]); */

}


///////////////////////////////////////////////////////////////////
// Part 1: Decrypt using ElGamal.
// This computes d_K = [y2 * (y1^a)^{-1}] mod p.
//
int decrypt_elgamal(int a, int p, int y1, int y2) {
  int y1exp = sqmult(a, p, y1);  // y1^a
  int *ii = compute_inverses(p, y1exp);  // compute (y1^a)^{-1}
  int y1expinv = *(ii+1);  // pick out (y1^a)^{-1}
  int d_K = (y2 * y1expinv) % p;  // decrypt

  //printf("y1, %d, y1exp = y1^a, %d, y1expinv, %d, check, %d, d_K, %d\n",
  //       y1, y1exp, y1expinv, (y1exp*y1expinv) % p, d_K);

  /* // Print output */
```

```
  /* printf("y1, %05d, y2, %05d, d_K, %05d\n", y1, y2, d_K); */
  /* num2str(d_K);  // print the plaintext */

  return d_K;
}


////////////////////////////////////////////////////////////////
// `compare` function for sorting
// Copied from https://stackoverflow.com/a/1791064
int compare_function(const void *a,const void *b) {
  int *x = (int *) a;
  int *y = (int *) b;
  return *x - *y;
}

////////////////////////////////////////////////////////////////
// Part 2: Shanks
//
// e_K(x,k) = (y1,y2) where y1 = alpha^k mod p and y2 = x*beta^k mod p
//    publicly known   : (p, alpha, beta)
//    private key      : a, which is known by Bob
//    chosen by Alice : k, which is used in encryption but *not* in decryption
//
// We use this code to find the k chosen by Alice.
//
// y1=3781 and y2=14409
//     y1 =  3781 = 5^k mod 31847
//     y2 = 14409 = (x*18074^k) mod 31847
//        x is the message (as a digit) that we now know
//
int shanks(int alpha, int beta, int p) {
  int m = ceil(sqrt(p));
  //printf("m, %d\n", m);

  // Initialize the lists
  int L1[m-1][2];
  int L2[m-1][2];

  // Compute the inverse of alpha first
  int *ii = compute_inverses(p, alpha);  // compute alpha^{-1}
  int alphainv = *(ii+1);  // pick out alpha^{-1}

  // Perform the loops
  for (int j=0; j<(m-1); j++) {
    //printf("m*j, %d\n", m*j);
    int r_alpha = sqmult(m*j, p, alpha);

```

```c
    // Put (r_alpha, j) into L1 [opposite of the notes, but it's fine]
    L1[j][0] = r_alpha;
    L1[j][1] = j;

    //printf("j, %d, r_alpha, %d\n", j, r_alpha);
  }

  for (int i=0; i<(m-1); i++) {
    // Compute (alpha^{-1})^i mod p
    int alphainv_to_i = sqmult(i, p, alphainv);

    // Compute beta * (alpha^{-1})^i mod p
    int r_beta = (beta * alphainv_to_i) % p;

    // Put (r_beta, i) into L2 [opposite of the notes, but it's fine]
    L2[i][0] = r_beta;
    L2[i][1] = i;

    //printf("i, %d, r_beta, %d\n", i, r_beta);
  }

  // Sort to create lists L1 and L2

  /* // Print the un-sorted array to check */
  /* printf("xxxxxxxxxxxxxxxx\nL1 un-sorted\n"); */
  /* for (int k=0; k<m-1; k++) */
  /*   printf("%d, %d\n", L1[k][0], L1[k][1]); */
  /* printf("xxxxxxxxxxxxxxxx\nL2 un-sorted\n"); */
  /* for (int k=0; k<m-1; k++) */
  /*   printf("%d, %d\n", L2[k][0], L2[k][1]); */

  // Sort the arrays in-place
  qsort(L1, sizeof(L1)/sizeof(*L1), sizeof(*L1), compare_function);
  qsort(L2, sizeof(L2)/sizeof(*L2), sizeof(*L2), compare_function);

  /* // Print the sorted arrays to check */
  /* printf("xxxxxxxxxxxxxxxx\nL1 sorted\n"); */
  /* for (int k=0; k<m-1; k++) */
  /*   printf("%d, %d\n", L1[k][0], L1[k][1]); */
  /* printf("xxxxxxxxxxxxxxxx\nL2 sorted\n"); */
  /* for (int k=0; k<m-1; k++) */
  /*   printf("%d, %d\n", L2[k][0], L2[k][1]); */
  /* printf("xxxxxxxxxxxxxxxx\n\n"); */


  // Search for identical coordinates in L1 and L2
  int i1 = 0;  // index into L1
  int i2 = 0;  // index into L2
```

```
  int j = 0;  // search for this
  int i = 0;  // search for this

  //for (int k=0; k<(m-1); k++) {
  int looking = 1;  // are we still looking?
  while ( (i1 < (m-1)) && (i2 < (m-1)) && looking ) {
    // Values
    int v1 = L1[i1][0];
    int v2 = L2[i2][0];

    //printf("v1, %d, v2, %d\n", v1, v2);

    // Logic to find v1 = v2
    if (v1 < v2)
      i1++;
    else if (v1 > v2)
      i2++;
    else {
      j = L1[i1][1];
      i = L2[i2][1];
      //printf("v1, %d, v2, %d\n", v1, v2);
      //printf(" j, %d,  i, %d\n", j, i);
      looking = 0;  // update the flag
    }
    //printf("v1, %02d, v2, %02d\n", v1, v2);
  }

  // Compute the answer a = log_alpha(beta) = (m*j + i) mod (p-1)
  int a = (m*j + i) % (p-1);

  /* // Check */
  /* // Find `a` that satisfies alpha^a = beta mod p */
  /* int r = sqmult(a, p, alpha); */
  /* printf("beta, %d, r, %d, (these should be equal)\n", beta, r); */


  // Return
  return a;
}


/////////////////////////////////////////////////////////////////
// Compute the values of k via brute force
//
int dl_brute_force (int alpha, int beta, int p) {

  int k=1;  // exponent value k
  int alphaexp = sqmult(k, p, alpha);
```

```

  while (alphaexp != beta) {
    k++;
    alphaexp = sqmult(k, p, alpha);
  }

  //printf("k, %d\n", k);
  return k;
}


///////////////////////////////////////////////////////////////////
// Functions to check

void check_num2str () {
  printf("\n\nchecking num2str...\n");
  num2str(2398);  // should be DOG
  num2str(1371);  // should be CAT
  num2str(17575);  // should be ZZZ
}

void check_inverses () {
  printf("\n\nchecking compute_inverses...\n");
  int r0 = 776;
  int r1 = 333;
  int *out = compute_inverses(r0, r1);
  printf("r0, 776, inverse should be 221, %d\n", *out);
  printf("r1, 333, inverse should be 261, %d\n", *(out+1));
}

void check_elgamal () {
  printf("\n\nchecking decrypt_elgamal...\n");
  int a  =  7899;
  int p  = 31847;
  int y1 =  3781;
  int y2 = 14409;
  decrypt_elgamal(a, p, y1, y2);
}

void run_elgamal () {
  printf("\n\nrunning decrypt_elgamal...\n");
  int a  =  7899;
  int p  = 31847;
  int y1;
  int y2;
  FILE *fptr;

  fptr = fopen("ElGamalcipher","r");
```

```
  for (int i=0; i<102; i++) {
    fscanf(fptr,"%d", &y1);
    fscanf(fptr,"%d", &y2);
    //printf("%d, %d\n", y1, y2);
    decrypt_elgamal(a, p, y1, y2);
  }
  fclose(fptr);
}

void run_shanks_tiny_test_case () {
  // Find `a` that satisfies alpha^a = beta mod p
  // Example:
  //     3^10 = 59049 = 8 mod 17
  //     int a = (m*j + i) % (p-1);
  //     m=5; j=2; i=0  ==>  10 % 16 (this is right!)

  printf("\n\nrunning shanks tiny test case...\n");

  int alpha =  3;
  int beta  =  8;
  int p     = 17;

  int a     = shanks(alpha, beta, p);

  printf("\n\nanswer `a` from Shanks, %d, (should be 10)\n\n\n", a);
}

void run_shanks () {
  // Find k that satisfies y1 = alpha^k mod p

  printf("\n\nrunning shanks...\n");

  //int a     =  7899;
  int p     = 31847;
  int alpha =     5;
  //int beta  = 18074;

  int y1;
  int y2;
  FILE *fptr;

  fptr = fopen("ElGamalcipher_first30","r");
  for (int i=0; i<30; i++) {
    fscanf(fptr,"%d", &y1);  // y1 = alpha^k mod p
    fscanf(fptr,"%d", &y2);  // y2 = (x*beta^k) mod p
    int beta = y1;  // rename for notational consistency
    int k = shanks(alpha, beta, p);
    printf("k%02d, %05d\n", i, k);
```

```
  }
}

void run_brute_force () {

  // Small values for testing
  //int alpha =  3;
  //int beta  =  8;
  //int p     = 17;

  int p     = 31847;
  int alpha =     5;

  int y1;
  int y2;
  FILE *fptr;

  fptr = fopen("ElGamalcipher_first30","r");
  for (int i=0; i<30; i++) {
    fscanf(fptr,"%d", &y1);  // y1 = alpha^k mod p
    fscanf(fptr,"%d", &y2);  // y2 = (x*beta^k) mod p
    int beta = y1;  // rename for notational consistency
    int k = dl_brute_force(alpha, beta, p);
    printf("brute-force k%02d, %05d\n", i, k);
  }

}


////////////////////////////////////////////////////////////////
// Call the functions here
//
int main() {
  // Check functions
  check_num2str();
  check_inverses();
  check_elgamal();

  // Run the decryption
  run_elgamal();
  run_shanks_tiny_test_case();
  run_shanks();
  run_brute_force();
}
```