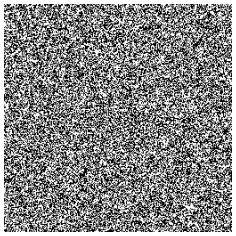
SuperCon'04 Problem Explanation

Decipher Encrypted Image

As announced in the problem outline, the subject of this year's problem is to compute the original from a given encrypted digital image. More specifically, you are given image data, consisting of 256 × 256 black(0)/white(1) pixels encrypted one by one, and your task is to decode this to get its original image. For example, given a figure of the left one below, you are asked to obtain the right one, by decrypting at each pixel. Since the decryption is extremely difficult, we will provide some additional information to help decryption.



Decryption
$\Longrightarrow$
$\Longleftarrow$
Encryption

The encrypted image and the hints are given by two files: `eimagefile` for the encrypted image, and `hintfile` for the hints.

Now the required task of the contest is precisely stated as follows.

Problem for SuperCon'04

Write a program that computes the original image or the one that is close to it based on given two files `eimagefile` and `hintfile`, the former for an encrypted image and the latter for hints.

The program must output a restored image (to the standard output) within three minutes. For the evaluation, each program is executed (for three minutes) on one pair of files `eimagefile` and `hintfile`, and the *closeness*, the proportion of pixels of the output that are the same as the original, is measured. Of course, the higher proportion is the better; but in case two programs give the same closeness, the tie is broken by choosing the one with shorter computation.

Note: Within the three-min. time limit, your program is allowed to produce images as many as you want. If there are more than one outputs, the last one is used for the evaluation.

In the following, we explain 1. our encryption method, 2. hints, 3. data formats and input/output, and 4. available tools.

1. Encryption Method

For the encryption, we use a standard RSA cryptosystem of 100 bit length. (See Appendix for basics on RSA cryptosystem.) For encrypting one image, one *RSA instance* (n, d, e) consisting of at most 100 bit numbers is used, where a pair of n and e is given as a public key in `hintfile`. With this public key, the encryption function e_{rsa} of this RSA instance is easily computable; on the other hand, it seems difficult (unless breaking this RSA instance) to compute the decryption function d_{rsa}.

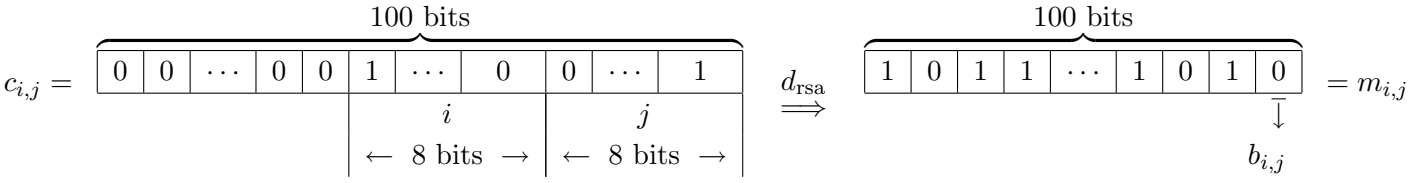
Note that these functions e_{rsa}, d_{rsa} are functions on the set $\{0, ..., n-1\}$. But since n is chosen as a 100 bit integer, you may simply consider them as functions on 100 bit integers.

Now consider any image, and let us see how it is encrypted by $(e_{\mathrm{rsa}}, d_{\mathrm{rsa}})$. Note that the image consists of 256×256 black and white pixels; assume here, for our explanation, that the image is given in an array `im`[256][256]. That is, for each (i, j), $0 \leq i, j \leq 255$, `im`$[i][j]$ is 0 (resp., 1) if the pixel located at the (i, j)th position, in the ith row from the top and the jth column from the left, is black (resp., white). On the other hand, an array `eim`[256][256] is used to keep the encrypted image. Then our encryption is stated as follows.

$$
\begin{array}{lcl}
c_{i,j} & = & i \times 256 + j, \\
m_{i,j} & = & d_{\mathrm{rsa}}(c_{i,j}), \\
b_{i,j} & = & \text{the rightmost bit (the least significant bit) of } m_{i,j}, \quad \text{and} \\
\texttt{eim}[i][j] & = & \texttt{im}[i][j] \oplus b_{i,j} \quad (\text{where } \oplus \text{ denotes the exclusive-or}).
\end{array}
$$

2. Hint Information

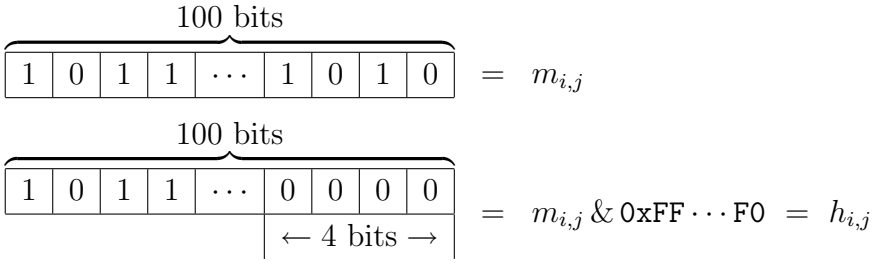
Intuitively, the decryption function d_{rsa} transforms a given 100 bit integer to some 100 integer, and the above encryption is illustrated as follows.



Thus, the decryption at each pixel is nothing but computing $b_{i,j}$ from its index (i, j). Clearly, this task is easy for those who *know* the secret key d because then d_{rsa} is easy to compute. The task, however, seems extremely difficult without knowing d.

Note, on the other hand, since you know how to compute e_{rsa}, it is still possible to compute $b_{i,j}$; you only need to try all candidates m for $m_{i,j}$ until the correct one yielding $c_{i,j} = e_{\mathrm{rsa}}(m)$ is found. But this is not feasible because you need to examine, in the worst case, 2^{100} candidates (precisely, n candidates). For reducing this hardness, we provide, for each (i, j)th pixel, a hint $h_{i,j}$ that is obtained from $m_{i,j}$ by erasing its last h bits.

For example, by masking the last $h = 4$ bits by 0, the following hint is obtained from $m_{i,j}$.



Given such a hint, the task is now to compute the masked or erased h bits; again, you can try all possible candidates. Note that it may be necessary, in the worst case, to compute e_{rsa} for all 2^h candidates; from this point of view, we call h a *hardness* of a hint.

3. Data Formats and Input/Output

An encrypted image data and the set of hints are respectively given in files `eimagefile` and `hintfile` in the following formats. Note that each hint $h_{i,j}$ is a 32 digit (128 bit) hexadecimal integers. (In general, for RSA computation, we will denote numbers by 32 digit hexadecimal integers.)

An example of the file `eimagefile`

```
0 1 1 0 1 0 1 ··· 1 0 1 1 0
0 1 0 0 0 1 0 ··· 0 0 0 0 1
  ⋮
```

∗ Each line gives the ith row of `eim`$[i][j]$ from the left, i.e., $j = 0$.

An example of the file `hintfile`

```
# hint for test data 1 (easy)
000000001BC8B211 4C4798F5F3E5A6EAF  # n (in 32 digit hexadecimal)
B # e (in one digit hexadecimal)
0  0  0000000000000000 0000000000000000  # hint for (0,0)
1  0  0000000000000000 0000000000000000  # hint for (0,1)
2 30  0000000007477FDB6 99D22F5A00000000  # hint for (0,2)
  ⋮
65535 15  00000000DB23889F C2EE8CC8E2018000  # hint for (255,255)
```

∗ The first line is for a comment, the second and the third lines are for the RSA public key. From the 4th line on, from index (i, j) with the smallest $c_{i,j}$, informaton for the (i, j)th pixel, i.e., a triple $c_{i,j}$, h, and $h_{i,j}$ is given in one line. In each line a comment may be given after `#`.

An answer, i.e., a (candidate for the) decrypted image, must be output to the standard output in the same format as the one for `eimagefile`. In fact, for recording the output

time, you must use the function `outresult` for the output. For using it, prepare an array `int image[65536]`, and put the decrypted image `im`$[i][j]$ into it in the following way, and call `outresult`. Then it yields the output in the desired format.

```
for(i = 0; i <= 255; i++)
   for(j = 0; j <= 255; j++) image[i*256+j] = im[i][j];
outresult(image)
```

Note that you may use `outresult` as many as possible within the time limit. When your program yields more than one image, we evaluate the last one produced before the time limit. The computation time is simply the time the program spend until the evaluated image from the beginning of its execution.

4. Available Tools

For your convenience, we provide some funtions useful for our problem. The prototypes of these functions are given in our common header file `supercon.h`, and their programs (given in `supercon.c`) are included by compiling your program with `supercon.o`. Our common header file also inlcudes some standard header files. You cannot include the other header files in your program.

Below we give a brief explanation on our common header file. For the details, read `supercon.h` and `supercon.c` by yourself.

header files that are included by supercon.h

`stdio.h`, `stdlib.h`, `string.h`, `mpi.h`

data types and and usefule functions

- type `bignum` for 128 bit integers

```
typedef struct bignum {
 unsigned long l;  (unsigned long means 64 bit integer)
 unsigned long h;
} bignum;
```

- modulo arithmetics of 128 bit integers

```
bignum madd(bignum x, bignum y, bignum n);      (x+y)%n
bignum msub(bignum x, bignum y, bignum n);      (x-y)%n
bignum mmul(bignum x, bignum y, bignum n);      (x*y)%n
```

- basic comparisons of 128 bit integers (`bool` is a type consisting of `false` (0) and `true` (1))

```
bool blt(bignum x, bignum y);     x < y (Less Than)
bool ble(bignum x, bignum y);     x <= y (Less than or Equal to)
bool bgt(bignum x, bignum y);     x > y (Greater Than)
bool bge(bignum x, bignum y);     x >= y (Greater than or Equal to)
bool beq(bignum x, bignum y);     x == y (EQual to)
```

`bool bne(bignum x, bignum y);` `x != y` (Not Equal to)

• basic arithmetics of 128 bit integers (only positive integers, overflow is ignored)

```
bignum badd(bignum x, bignum y); x + y
bignum bsub(bignum x, bignum y); x − y
bignum bmul(bignum x, bignum y); x * y
bignum bquo(bignum x, bignum y); x / y
bignum brem(bignum x, bignum y); x % y
void bdiv(bignum x, bignum y, bignum * q, bignum * r);
```

• data input/output

```
int getimage(FILE *fp, int eimage[]);
int gethint(FILE *fp, bignum *rsaN, bignum *rsaE, int hardness[], hint[]);
void outstarttime(void);      command for evaluation (must execute first)
void outresult(int image[]); output an answer image (must use this for output)
```

Note

1. `getimage`, `gethint` respectively returns `1` for the normal case, and returns `0` if it fails, and
2. each array must be defined as follows (where `Nxy` is the total number of pixels, i.e., 65536).

```
int image[Nxy], eimage[Nxy];
int hardness[Nxy]; bignum hint[Nxy];
```

• input/output 128 bit integers (for debugging)

```
void getnum16(bignum *x);     get num. in hexadecimal to x
void getnum10(bignum *x);     get num. in decimal to x
void putnum16(bignum x);      print x in hexadecimal
void putnum10(bignum x);      print x in decimal
bignum btransnum(unsigned long h, unsigned long l);
                              transform 19+19 digit integer in decimal to bignum
```

∗ max. `bignum` is `3402823669209384634 8034375210639556607` (38 digits in decimal)

Note that `supercon.o` is complied from `supercon.c` with the strongest optimized mode. You are not allowed to make any change on `supercon.h`, `supercon.c`, or `supercon.o`. But, except for `outstarttime` and `outresult`, you may define your own functions relpacing the above functions and use them in your program. (Such functions must be defined in your program.)

Appendix: RSA Cryptosystem(Very Basics Only)

RSA Cryptosystem is one of the public key cryptosystems, and it is a way to convert a *plain text* encoded as an integer to a *cipher text* also encoded as an integer.

One *RSA instance* (or *RSA set*) of RSA Cryptosystem is specified by a triple (n, d, e) of positive integers. A pair (n, e) is called an *encryption key* or a *public key*, and d is called a *decription key* or a *secret key*. These three numbers are defined as follows from two big (secret) prime numbers p and q.

$$\begin{aligned} n &= p \times q, \\ \phi &= (p-1) \times (q-1), \quad \text{and} \\ d \times e &\equiv 1 \pmod{\phi}. \end{aligned}$$

This relationship does not determine a pair d and e. But for our problem, we will fix $e = 11$ (= `0xB`); then d is uniquely determined under modulo n. We choose p and q from prime numbers of at most 50 bits, you may assume that n is a 100 bit integer (at most).

For each RSA instance, we can define an *encryption function*, a function computing a cipher text, and a *decryption function*, a function computing a plain text, in the following way.

$$\begin{aligned} \text{encryption function:} \quad e_{\text{rsa}}(m) &= m^e \pmod{n}, \quad \text{and} \\ \text{decryption function:} \quad d_{\text{rsa}}(c) &= c^d \pmod{n}. \end{aligned}$$

Clearly, we have $d_{\text{rsa}}(e_{\text{rsa}}(m)) = m$; but notice that the relation $e_{\text{rsa}}(d_{\text{rsa}}(c)) = c$ also holds.

Breaking RSA Cryptosystem

Usually, the public key (n, e) of an RSA instance (n, d, e) is made public, and p, q, and d are kept secret. If someone can factorize n into p and q, then he/she can easily compute d, and thereby, computing d_{rsa} efficiently by using d. If such a situation happens, then we say that the RSA instance (n, d, e) is *broken*.

Weak Point of RSA!? (Useful?! for Solving Our Problem)

A decrypting function d_{rsa} has the following property; by using this, if both c_1 and c_2 are decrypted, then we can also decrypt $c_3 = c_1 \times c_2$.

$$\left.\begin{aligned} d_{\text{rsa}}(c_1) &= m_1 \\ d_{\text{rsa}}(c_2) &= m_2 \end{aligned}\right\} \Rightarrow d_{\text{rsa}}(c_1 \times c_2) \equiv m_1 \times m_2 \bmod n.$$

Usually, this may not cause any serious problem, but this may create a weak point of a cryptosystem if RSA cryptosystem is used not in an appropriate manner. In fact, you may be able to use this property to design an efficient decoding program for our problem!