History of Cryptography

The art of cryptography is considered to be born along with the art of writing. As civilizations evolved, human beings got organized in tribes, groups, and kingdoms. This led to the emergence of ideas such as power, battles, supremacy, and politics. These ideas further fueled the natural need of people to communicate secretly with selective recipient which in turn ensured the continuous evolution of cryptography as well.

The roots of cryptography are found in Roman and Egyptian civilizations.

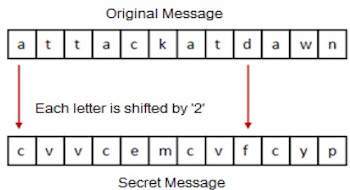
Hieroglyph − The Oldest Cryptographic Technique

The first known evidence of cryptography can be traced to the use of ‘hieroglyph’. Some 4000 years ago, the Egyptians used to communicate by messages written in hieroglyph. This code was the secret known only to the scribes who used to transmit messages on behalf of the kings. One such hieroglyph is shown below.



Later, the scholars moved on to using simple mono-alphabetic substitution ciphers during 500 to 600 BC. This involved replacing alphabets of message with other alphabets with some secret rule. This **rule** became a **key** to retrieve the message back from the garbled message.

The earlier Roman method of cryptography, popularly known as the **Caesar Shift Cipher,** relies on shifting the letters of a message by an agreed number (three was a common choice), the recipient of this message would then shift the letters back by the same number and obtain the original message.



## Monoalphabetic Cipher

Monoalphabetic cipher is a substitution cipher in which for a given key, the cipher alphabet for each plain alphabet is fixed throughout the encryption process. For example, if ‘A’ is encrypted as ‘D’, for any number of occurrence in that plaintext, ‘A’ will always get encrypted to ‘D’.

All of the substitution ciphers we have discussed earlier in this chapter are monoalphabetic; these ciphers are highly susceptible to cryptanalysis.

Code:

#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

char s[30],k[27],c[30];

int i, index;

clrscr();

printf("Enter plain text: ");

gets(s);

printf("\nEnter key with 26 character:");

for(i=0;i<26;i++)

{

printf("\n%c",i+97);

k[i]=getch();

printf("%c",k[i]);

}

for(i=0;i<strlen(s);i++)

{

index=s[i]-97;

c[i]=k[index];

}

printf("Your cipher text is:");

for(i=0;i<strlen(s);i++)

{

printf("%c",c[i]);

}

getch();

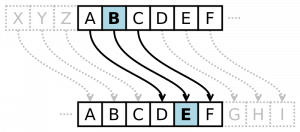
}

Caesar Cipher

It is one of the simplest encryption technique in which each character in plain text is replaced by a character some fixed number of positions down to it.

For example, if key is 3 then we have to replace character by another character that is 3 position down to it. Like A will be replaced by D, C will be replaced by F and so on.

For decryption just follow the reverse of encryption process.



Encryption

#include<stdio.h>

int main()

{

    char message[100], ch;

    int i, key;

    printf("Enter a message to encrypt: ");

    gets(message);

    printf("Enter key: ");

    scanf("%d", &key);

    for(i = 0; message[i] != '\0'; ++i){

        ch = message[i];

        if(ch >= 'a' && ch <= 'z'){

            ch = ch + key;

            if(ch > 'z'){

                ch = ch - 'z' + 'a' - 1;

            }

            message[i] = ch;

        }

        else if(ch >= 'A' && ch <= 'Z'){

            ch = ch + key;

            if(ch > 'Z'){

                ch = ch - 'Z' + 'A' - 1;

            }

            message[i] = ch;

        }

    }

    printf("Encrypted message: %s", message);

    return 0;

}

Decryption

#include<stdio.h>

int main()

{

    char message[100], ch;

    int i, key;

    printf("Enter a message to decrypt: ");

    gets(message);

    printf("Enter key: ");

    scanf("%d", &key);

    for(i = 0; message[i] != '\0'; ++i){

        ch = message[i];

        if(ch >= 'a' && ch <= 'z'){

            ch = ch - key;

            if(ch < 'a'){

                ch = ch + 'z' - 'a' + 1;

            }

            message[i] = ch;

        }

        else if(ch >= 'A' && ch <= 'Z'){

            ch = ch - key;

            if(ch < 'A'){

                ch = ch + 'Z' - 'A' + 1;

            }

            message[i] = ch;

        }

    }

    printf("Decrypted message: %s", message);

    return 0;

}

Simple Substitution Cipher

It is an improvement to the Caesar Cipher. Instead of shifting the alphabets by some number, this scheme uses some permutation of the letters in alphabet.

For example, A.B…..Y.Z and Z.Y……B.A are two obvious permutation of all the letters in alphabet. Permutation is nothing but a jumbled up set of alphabets.

With 26 letters in alphabet, the possible permutations are 26! (Factorial of 26) which is equal to 4x1026. The sender and the receiver may choose any one of these possible permutation as a ciphertext alphabet. This permutation is the secret key of the scheme.

Process of Simple Substitution Cipher

* Write the alphabets A, B, C,...,Z in the natural order.
* The sender and the receiver decide on a randomly selected permutation of the letters of the alphabet.
* Underneath the natural order alphabets, write out the chosen permutation of the letters of the alphabet. For encryption, sender replaces each plaintext letters by substituting the permutation letter that is directly beneath it in the table. This process is shown in the following illustration. In this example, the chosen permutation is K,D, G, ..., O. The plaintext ‘point’ is encrypted to ‘MJBXZ’.

Here is a jumbled Ciphertext alphabet, where the order of the ciphertext letters is a key.

Simple Substitution Cipher

* On receiving the ciphertext, the receiver, who also knows the randomly chosen permutation, replaces each ciphertext letter on the bottom row with the corresponding plaintext letter in the top row. The ciphertext ‘MJBXZ’ is decrypted to ‘point’.

Security Value

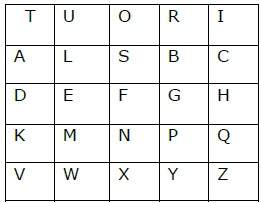
Simple Substitution Cipher is a considerable improvement over the Caesar Cipher. The possible number of keys is large (26!) and even the modern computing systems are not yet powerful enough to comfortably launch a brute force attack to break the system. However, the Simple Substitution Cipher has a simple design and it is prone to design flaws, say choosing obvious permutation, this cryptosystem can be easily broken.

Playfair Cipher

In this scheme, pairs of letters are encrypted, instead of single letters as in the case of simple substitution cipher.

In playfair cipher, initially a key table is created. The key table is a 5×5 grid of alphabets that acts as the key for encrypting the plaintext. Each of the 25 alphabets must be unique and one letter of the alphabet (usually J) is omitted from the table as we need only 25 alphabets instead of 26. If the plaintext contains J, then it is replaced by I.

The sender and the receiver deicide on a particular key, say ‘tutorials’. In a key table, the first characters (going left to right) in the table is the phrase, excluding the duplicate letters. The rest of the table will be filled with the remaining letters of the alphabet, in natural order. The key table works out to be −



Process of Playfair Cipher

* First, a plaintext message is split into pairs of two letters (digraphs). If there is an odd number of letters, a Z is added to the last letter. Let us say we want to encrypt the message “hide money”. It will be written as −

HI DE MO NE YZ

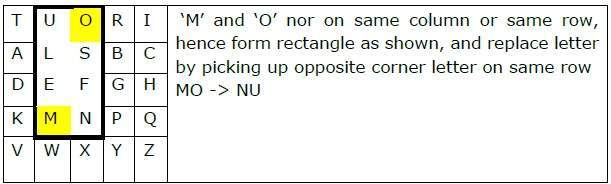
* The rules of encryption are −
  + If both the letters are in the same column, take the letter below each one (going back to the top if at the bottom)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| T | U | O | R | I | ‘H’ and ‘I’ are in same column, hence take letter below them to replace. HI → QC |
| A | L | S | B | C |
| D | E | F | G | H |
| K | M | N | P | Q |
| V | W | X | Y | Z |

* If both letters are in the same row, take the letter to the right of each one (going back to the left if at the farthest right)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| T | U | O | R | I | ‘D’ and ‘E’ are in same row, hence take letter to the right of them to replace. DE → EF |
| A | L | S | B | C |
| D | E | F | G | H |
| K | M | N | P | Q |
| V | W | X | Y | Z |

* If neither of the preceding two rules are true, form a rectangle with the two letters and take the letters on the horizontal opposite corner of the rectangle.



Using these rules, the result of the encryption of ‘hide money’ with the key of ‘tutorials’ would be −

QC EF NU MF ZV

Decrypting the Playfair cipher is as simple as doing the same process in reverse. Receiver has the same key and can create the same key table, and then decrypt any messages made using that key.

Security Value

It is also a substitution cipher and is difficult to break compared to the simple substitution cipher. As in case of substitution cipher, cryptanalysis is possible on the Playfair cipher as well, however it would be against 625 possible pairs of letters (25x25 alphabets) instead of 26 different possible alphabets.

The Playfair cipher was used mainly to protect important, yet non-critical secrets, as it is quick to use and requires no special equipment.

Vigenere Cipher

This scheme of cipher uses a text string (say, a word) as a key, which is then used for doing a number of shifts on the plaintext.

For example, let’s assume the key is ‘point’. Each alphabet of the key is converted to its respective numeric value: In this case,

p → 16, o → 15, i → 9, n → 14, and t → 20.

Thus, the key is: 16 15 9 14 20.

Process of Vigenere Cipher

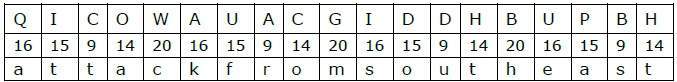
* The sender and the receiver decide on a key. Say ‘point’ is the key. Numeric representation of this key is ‘16 15 9 14 20’.
* The sender wants to encrypt the message, say ‘attack from south east’. He will arrange plaintext and numeric key as follows −

Vigenere Cipher

* He now shifts each plaintext alphabet by the number written below it to create ciphertext as shown below −



* Here, each plaintext character has been shifted by a different amount – and that amount is determined by the key. The key must be less than or equal to the size of the message.
* For decryption, the receiver uses the same key and shifts received ciphertext in reverse order to obtain the plaintext.



Security Value

Vigenere Cipher was designed by tweaking the standard Caesar cipher to reduce the effectiveness of cryptanalysis on the ciphertext and make a cryptosystem more robust. It is significantly **more secure than a regular Caesar Cipher**.

In the history, it was regularly used for protecting sensitive political and military information. It was referred to as the **unbreakable cipher** due to the difficulty it posed to the cryptanalysis.

Variants of Vigenere Cipher

There are two special cases of Vigenere cipher −

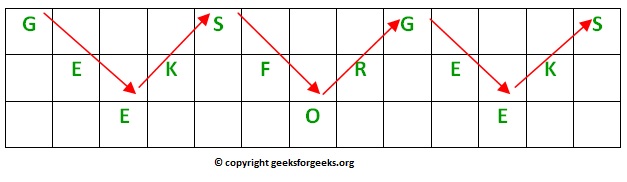
* The keyword length is same as plaintect message. This case is called **Vernam Cipher**. It is more secure than typical Vigenere cipher.
* Vigenere cipher becomes a cryptosystem with perfect secrecy, which is called **One-time pad**.

Rail Fence Cipher

**Encryption**

In a transposition cipher, the order of the alphabets is re-arranged to obtain the cipher-text.

* In the rail fence cipher, the plain-text is written downwards and diagonally on successive rails of an imaginary fence.
* When we reach the bottom rail, we traverse upwards moving diagonally, after reaching the top rail, the direction is changed again. Thus the alphabets of the message are written in a zig-zag manner.
* After each alphabet has been written, the individual rows are are combined to obtain the cipher-text.

For example, if the message is “GeeksforGeeks” and the number of rails = 3 then cipher is prepared as:  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Untitled1.jpg)

**Decryption**

As we’ve seen earlier, the number of columns in rail fence cipher remains equal to the length of plain-text message. And the key corresponds to the number of rails.

* Hence, rail matrix can be constructed accordingly. Once we’ve got the matrix we can figure-out the spots where texts should be placed (using the same way of moving diagonally up and down alternatively ).
* Then, we fill the cipher-text row wise. After filling it, we traverse the matrix in zig-zag manner to obtain the original text.

## Vernam Cipher

As introduction to stream ciphers, and to demonstrate that a perfect cipher does exist, we describe the Vernam Cipher, also known as the one-time-pad.

Gilbert Vernam invented and patented his cipher in 1917 while working at AT&T. The teletype had been recently introduced, and along with this the commerical Baudot code. Now messages were uniformly thought of as streams of zero's and one's (But the word "bit" was not yet invented. This is due to Shannon in the 40's.)

Vernam proposed a bit-wise exclusive or of the message stream with a truely random zero-one stream which was shared by sender and receipient.

**Example:**

SENDING

-------

message: 0 0 1 0 1 1 0 1 0 1 1 1 ...

pad: 1 0 0 1 1 1 0 0 1 0 1 1 ...

XOR ---------------------------

cipher: 1 0 1 1 0 0 0 1 1 1 0 0 ...

RECEIVING

---------

cipher: 1 0 1 1 0 0 0 1 1 1 0 0 ...

pad: 1 0 0 1 1 1 0 0 1 0 1 1 ...

XOR ---------------------------

message: 0 0 1 0 1 1 0 1 0 1 1 1 ...

This cipher is unbreakable in a very strong sense. The intuition is that any message can be transformed into any cipher (of the same length) by a pad, and all transformations are equally likely. Given a two letter message, there is a pad which adds to the message to give OK, and another pad which adds to the message to give NO. Since either of these pads are equally likely, the message is equally likely to be OK or NO.

# RSA Algorithm

RSA algorithm is asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e. **Public Key** and **Private Key.** As the name describes that the Public Key is given to everyone and Private key is kept private.

**An example of asymmetric cryptography :**

1. A client (for example browser) sends its public key to the server and requests for some data.
2. The server encrypts the data using client’s public key and sends the encrypted data.
3. Client receives this data and decrypts it.

Since this is asymmetric, nobody else except browser can decrypt the data even if a third party has public key of browser.

**The idea!** The idea of RSA is based on the fact that it is difficult to factorize a large integer. The public key consists of two numbers where one number is multiplication of two large prime numbers. And private key is also derived from the same two prime numbers. So if somebody can factorize the large number, the private key is compromised. Therefore encryption strength totally lies on the key size and if we double or triple the key size, the strength of encryption increases exponentially. RSA keys can be typically 1024 or 2048 bits long, but experts believe that 1024 bit keys could be broken in the near future. But till now it seems to be an infeasible task.

**Let us learn the mechanism behind RSA algorithm :**

**>> Generating Public Key :**

* Select two prime no's. Suppose **P = 53 and Q = 59**.
* Now First part of the Public key : **n = P\*Q = 3127**.
* We also need a small exponent say **e** :
* But e Must be
  + An integer.
  + Not be a factor of n.
* + **1 < e <** [**Φ(n)**](http://www.geeksforgeeks.org/eulers-totient-function/) [Φ(n) is discussed below],
  + Let us now consider it to be equal to 3.

* Our Public Key is made of n and e

**>> Generating Private Key :**

* We need to calculate Φ(n) :
* Such that **Φ(n) = (P-1)(Q-1)**
* so, Φ(n) = 3016

* Now calculate Private Key, **d** :
* **d = (k\*Φ(n) + 1) / e** for some integer k
* For k = 2, value of d is 2011.

Now we are ready with our – Public Key ( n = 3127 and e = 3) and Private Key(d = 2011)

Now we will encrypt **“HI”** :

* Convert letters to numbers : H = 8 and I = 9

* Thus **Encrypted Data c = 89e mod n**.
* Thus our Encrypted Data comes out to be 1394

Now we will decrypt **1349** :

* **Decrypted Data = cd mod n**.
* Thus our Encrypted Data comes out to be 89

**8 = H and I = 9 i.e. "HI".**

**Below is C implementation of RSA algorithm for small values:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| // C program for RSA asymmetric cryptographic  // algorithm. For demonstration values are  // relatively small compared to practical  // application  #include<stdio.h>  #include<math.h>    // Returns gcd of a and b  int gcd(int a, int h)  {      int temp;      while (1)      {          temp = a%h;          if (temp == 0)            return h;          a = h;          h = temp;      }  }    // Code to demonstrate RSA algorithm  int main()  {      // Two random prime numbers      double p = 3;      double q = 7;        // First part of public key:      double n = p\*q;        // Finding other part of public key.      // e stands for encrypt      double e = 2;      double phi = (p-1)\*(q-1);      while (e < phi)      {          // e must be co-prime to phi and          // smaller than phi.          if (gcd(e, phi)==1)              break;          else              e++;      }        // Private key (d stands for decrypt)      // choosing d such that it satisfies      // d\*e = 1 + k \* totient      int k = 2;  // A constant value      double d = (1 + (k\*phi))/e;        // Message to be encrypted      double msg = 20;        printf("Message data = %lf", msg);        // Encryption c = (msg ^ e) % n      double c = pow(msg, e);      c = fmod(c, n);      printf("\nEncrypted data = %lf", c);        // Decryption m = (c ^ d) % n      double m = pow(c, d);      m = fmod(m, n);      printf("\nOriginal Message Sent = %lf", m);        return 0;  }  // This code is contributed by Akash Sharan.  Diffie-Hellman  Diffie-Hellman is a way of generating a shared secret between two people in such a way that the secret can't be seen by observing the communication. That's an important distinction: **You're not sharing information during the key exchange, you're creating a key together.**  The basic idea works like this:   1. I come up with two prime numbers **g** and **p** and tell you what they are. 2. You then pick a secret number (**a**), but you don't tell anyone. Instead you compute **ga** *mod* **p**and send that result back to me. (We'll call that **A** since it came from **a**). 3. I do the same thing, but we'll call my secret number **b** and the computed number **B**. So I compute **gb** *mod* **p** and send you the result (called "**B**") 4. Now, you take the number I sent you and do the exact same operation with *it*. So that's **Ba** *mod***p**. 5. I do the same operation with the result you sent me, so: **Ab** *mod* **p**.   The "magic" here is that the answer I get at step 5 is *the same number* you got at step 4. Now it's not really magic, it's just math, and it comes down to a fancy property of modulo exponents. Specifically:  **(ga** *mod* **p)b** *mod* **p** = **gab** *mod* **p** **(gb** *mod* **p)a** *mod* **p** = **gba** *mod* **p**  **Knapsack Algorithm**  The First General Public-Key Algorithm used what we call the Knapsack Algorithm. Although we now know that this algorithm is not secure we can use it to look at how these types of encryption mechanisms work.  The knapsack algorithm works like this: Imagine you have a set of different weights which you can use to make any total weight that you need by adding combinations of any of these weights together. Let us look at an example: Imagine you had a set of weights 1, 6, 8, 15 and 24. To pack a knapsack weighing 30, you could use weights 1, 6, 8 and 15. It would not be possible to pack a knapsack that weighs 17 but this might not matter. You might represent the weight 30 by the binary code 11110 (one 1, one 6, one 8, one 15 and no 24). Example:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Plain text | 10011 | 11010 | 01011 | 00000 | | Knapsack | 1 6 8 15 24 | 1 6 8 15 24 | 1 6 8 15 24 | 1 6 8 15 24 | | Cipher text | 1 + 15 + 24 = 40 | 1 + 6 + 15 = 22 | 6 + 15 + 24 = 45 | 0 = 0 |   What total weights is it possible to make?  So, if someone sends you the code 38 this can only have come from the plain text 01101. When the Knapsack Algorithm is used in public key cryptography, the idea is to create two different knapsack problems. One is easy to solve, the other not. Using the easy knapsack, the hard knapsack is derived from it. The hard knapsack becomes the public key. The easy knapsack is the private key. The public key can be used to encrypt messages, but cannot be used to decrypt messages. The private key decrypts the messages.   The Superincreasing Knapsack Problem An easy knapsack problem is one in which the weights are in a superincreasing sequence. A superincreasing sequence is one in which the next term of the sequence is greater than the sum of all preceding terms. For example, the set {1, 2, 4, 9, 20, 38} is superincreasing, but the set {1, 2, 3, 9, 10, 24} is not because 10 < 1+2+3+9.  It is easy to solve a superincreasing knapsack. Simply take the total weight of the knapsack and compare it with the largest weight in the sequence. If the total weight is less than the number, then it is not in the knapsack. If the total weight is greater then the number, it is in the knapsack. Subtract the number from the total, and compare with the next highest number. Keep working this way until the total reaches zero. If the total doesn't reach zero, then there is no solution.  So, for example, if you have a knapsack that weighs 23 that has been made from the weights of the superincreasing series {1, 2, 4, 9, 20, 38} then it does not contain the weight 38 (as 38 > 23) but it does contain the weight 20; leaving 3; which does not contain the weight 9 still leaving 3; which does not contain the weight 4 still leaving 3; which contains the weight 2, leaving 1; which contains the weight 1. The binary code is therefore 110010.  It is much harder to decrypt a non-superincreasing knapsack problem. Give a friend a non-superincreasing knapsack and a total and see why this is the case. One algorithm that uses a superincreasing knapsack for the private (easy) key and a non-superincreasing knapsack for the public key was created by Merkle and Hellman They did this by taking a superincreasing knapsack problem and converting it into a non-superincreasing one that could be made public, using modulus arithmetic.  Making the Public Key  To produce a normal knapsack sequence, take a superincreasing sequence; e.g. {1, 2, 4, 10, 20, 40}. Multiply all the values by a number, n, modulo m. The modulus should be a number greater than the sum of all the numbers in the sequence, for example, 110. The multiplier should have no factors in common with the modulus. So let's choose 31. The normal knapsack sequence would be:  1×31 mod(110) = 31 2×31 mod(110) = 62 4×31 mod(110) = 14 10×31 mod(110) = 90 20×31 mod(110) = 70 40×31 mod(110) = 30  So the public key is: {31, 62, 14, 90, 70, 30} and the private key is {1, 2, 4, 10, 20.40}.  Let's try to send a message that is in binary code: 100100111100101110 The knapsack contains six weights so we need to split the message into groups of six: 100100 111100 101110 This corresponds to three sets of weights with totals as follows 100100 = 31 + 90 = 121 111100 = 31+62+14+90 = 197 101110 = 31+14+90+70 =205 So the coded message is 121 197 205.  Now the receiver has to decode the message... The person decoding must know the two numbers 110 and 31 (the modulus and the multiplier). Let's call the modulus "m" and the number you multiply by "n". We need *n*−1, which is a multiplicative inverse of n mod m, i.e. *n*(*n*−1) = 1 mod m  In this case I have calculated *n*−1 to be 71.    All you then have to do is multiply each of the codes 71 mod 110 to find the total in the knapsack which contains {1, 2, 4, 10, 20, 40} and hence to decode the message. The coded message is 121 197 205:  121×71 mod(110) = 11 = 100100 197×71 mod(110) = 17 = 111100 205×71 mod(110) = 35 = 101110  The decoded message is: 100100111100101110. Just as I thought!  Simple and short knapsack codes are far too easy to break to be of any real use. For a knapsack code to be reasonably secure it would need well over 200 terms each of length 200 bits. Data Encryption Standard The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST).  DES is an implementation of a Feistel Cipher. It uses 16 round Feistel structure. The block size is 64-bit. Though, key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only). General Structure of DES is depicted in the following illustration −  DES Structure  Since DES is based on the Feistel Cipher, all that is required to specify DES is −   * Round function * Key schedule * Any additional processing − Initial and final permutation   Initial and Final Permutation  The initial and final permutations are straight Permutation boxes (P-boxes) that are inverses of each other. They have no cryptography significance in DES. The initial and final permutations are shown as follows −  Initial and Final Permutation  Round Function  The heart of this cipher is the DES function, *f*. The DES function applies a 48-bit key to the rightmost 32 bits to produce a 32-bit output.  Round Function   * **Expansion Permutation Box** − Since right input is 32-bit and round key is a 48-bit, we first need to expand right input to 48 bits. Permutation logic is graphically depicted in the following illustration −   Permutation Logic   * The graphically depicted permutation logic is generally described as table in DES specification illustrated as shown −   DES Specification   * **XOR (Whitener).** − After the expansion permutation, DES does XOR operation on the expanded right section and the round key. The round key is used only in this operation. * **Substitution Boxes.** − The S-boxes carry out the real mixing (confusion). DES uses 8 S-boxes, each with a 6-bit input and a 4-bit output. Refer the following illustration −   S-boxes   * The S-box rule is illustrated below −   S-box Rule   * There are a total of eight S-box tables. The output of all eight s-boxes is then combined in to 32 bit section. * **Straight Permutation** − The 32 bit output of S-boxes is then subjected to the straight permutation with rule shown in the following illustration:   Straight Permutation  Key Generation  The round-key generator creates sixteen 48-bit keys out of a 56-bit cipher key. The process of key generation is depicted in the following illustration −  Key Generation  The logic for Parity drop, shifting, and Compression P-box is given in the DES description.  DES Analysis  The DES satisfies both the desired properties of block cipher. These two properties make cipher very strong.   * **Avalanche effect** − A small change in plaintext results in the very grate change in the ciphertext. * **Completeness** − Each bit of ciphertext depends on many bits of plaintext.   During the last few years, cryptanalysis have found some weaknesses in DES when key selected are weak keys. These keys shall be avoided.  DES has proved to be a very well designed block cipher. There have been no significant cryptanalytic attacks on DES other than exhaustive key search. Cryptography Hash functions Hash functions are extremely useful and appear in almost all information security applications.  A hash function is a mathematical function that converts a numerical input value into another compressed numerical value. The input to the hash function is of arbitrary length but output is always of fixed length.  Values returned by a hash function are called **message digest** or simply **hash values**. The following picture illustrated hash function −  Hash Functions  Features of Hash Functions  The typical features of hash functions are −   * **Fixed Length Output (Hash Value)**   + Hash function coverts data of arbitrary length to a fixed length. This process is often referred to as **hashing the data**.   + In general, the hash is much smaller than the input data, hence hash functions are sometimes called **compression functions**.   + Since a hash is a smaller representation of a larger data, it is also referred to as a **digest**.   + Hash function with n bit output is referred to as an **n-bit hash function**. Popular hash functions generate values between 160 and 512 bits. * **Efficiency of Operation**   + Generally for any hash function h with input x, computation of h(x) is a fast operation.   + Computationally hash functions are much faster than a symmetric encryption.   Properties of Hash Functions  In order to be an effective cryptographic tool, the hash function is desired to possess following properties −   * **Pre-Image Resistance**   + This property means that it should be computationally hard to reverse a hash function.   + In other words, if a hash function h produced a hash value z, then it should be a difficult process to find any input value x that hashes to z.   + This property protects against an attacker who only has a hash value and is trying to find the input. * **Second Pre-Image Resistance**   + This property means given an input and its hash, it should be hard to find a different input with the same hash.   + In other words, if a hash function h for an input x produces hash value h(x), then it should be difficult to find any other input value y such that h(y) = h(x).   + This property of hash function protects against an attacker who has an input value and its hash, and wants to substitute different value as legitimate value in place of original input value. * **Collision Resistance**   + This property means it should be hard to find two different inputs of any length that result in the same hash. This property is also referred to as collision free hash function.   + In other words, for a hash function h, it is hard to find any two different inputs x and y such that h(x) = h(y).   + Since, hash function is compressing function with fixed hash length, it is impossible for a hash function not to have collisions. This property of collision free only confirms that these collisions should be hard to find.   + This property makes it very difficult for an attacker to find two input values with the same hash.   + Also, if a hash function is collision-resistant **then it is second pre-image resistant.**   Design of Hashing Algorithms  At the heart of a hashing is a mathematical function that operates on two fixed-size blocks of data to create a hash code. This hash function forms the part of the hashing algorithm.  The size of each data block varies depending on the algorithm. Typically the block sizes are from 128 bits to 512 bits. The following illustration demonstrates hash function −  Hash Function Structure  Hashing algorithm involves rounds of above hash function like a block cipher. Each round takes an input of a fixed size, typically a combination of the most recent message block and the output of the last round.  This process is repeated for as many rounds as are required to hash the entire message. Schematic of hashing algorithm is depicted in the following illustration −  Hashing Algorithm  Since, the hash value of first message block becomes an input to the second hash operation, output of which alters the result of the third operation, and so on. This effect, known as an **avalanche** effect of hashing.  Avalanche effect results in substantially different hash values for two messages that differ by even a single bit of data.  Understand the difference between hash function and algorithm correctly. The hash function generates a hash code by operating on two blocks of fixed-length binary data.  Hashing algorithm is a process for using the hash function, specifying how the message will be broken up and how the results from previous message blocks are chained together.  Popular Hash Functions  Let us briefly see some popular hash functions −  Message Digest (MD)  MD5 was most popular and widely used hash function for quite some years.   * The MD family comprises of hash functions MD2, MD4, MD5 and MD6. It was adopted as Internet Standard RFC 1321. It is a 128-bit hash function. * MD5 digests have been widely used in the software world to provide assurance about integrity of transferred file. For example, file servers often provide a pre-computed MD5 checksum for the files, so that a user can compare the checksum of the downloaded file to it. * In 2004, collisions were found in MD5. An analytical attack was reported to be successful only in an hour by using computer cluster. This collision attack resulted in compromised MD5 and hence it is no longer recommended for use.   Secure Hash Function (SHA)  Family of SHA comprise of four SHA algorithms; SHA-0, SHA-1, SHA-2, and SHA-3. Though from same family, there are structurally different.   * The original version is SHA-0, a 160-bit hash function, was published by the National Institute of Standards and Technology (NIST) in 1993. It had few weaknesses and did not become very popular. Later in 1995, SHA-1 was designed to correct alleged weaknesses of SHA-0. * SHA-1 is the most widely used of the existing SHA hash functions. It is employed in several widely used applications and protocols including Secure Socket Layer (SSL) security. * In 2005, a method was found for uncovering collisions for SHA-1 within practical time frame making long-term employability of SHA-1 doubtful. * SHA-2 family has four further SHA variants, SHA-224, SHA-256, SHA-384, and SHA-512 depending up on number of bits in their hash value. No successful attacks have yet been reported on SHA-2 hash function. * Though SHA-2 is a strong hash function. Though significantly different, its basic design is still follows design of SHA-1. Hence, NIST called for new competitive hash function designs. * In October 2012, the NIST chose the Keccak algorithm as the new SHA-3 standard. Keccak offers many benefits, such as efficient performance and good resistance for attacks.   RIPEMD  The RIPEND is an acronym for RACE Integrity Primitives Evaluation Message Digest. This set of hash functions was designed by open research community and generally known as a family of European hash functions.   * The set includes RIPEND, RIPEMD-128, and RIPEMD-160. There also exist 256, and 320-bit versions of this algorithm. * Original RIPEMD (128 bit) is based upon the design principles used in MD4 and found to provide questionable security. RIPEMD 128-bit version came as a quick fix replacement to overcome vulnerabilities on the original RIPEMD. * RIPEMD-160 is an improved version and the most widely used version in the family. The 256 and 320-bit versions reduce the chance of accidental collision, but do not have higher levels of security as compared to RIPEMD-128 and RIPEMD-160 respectively.   Whirlpool  This is a 512-bit hash function.   * It is derived from the modified version of Advanced Encryption Standard (AES). One of the designer was Vincent Rijmen, a co-creator of the AES. * Three versions of Whirlpool have been released; namely WHIRLPOOL-0, WHIRLPOOL-T, and WHIRLPOOL.   Applications of Hash Functions  There are two direct applications of hash function based on its cryptographic properties.  Password Storage  Hash functions provide protection to password storage.   * Instead of storing password in clear, mostly all logon processes store the hash values of passwords in the file. * The Password file consists of a table of pairs which are in the form (user id, h(P)). * The process of logon is depicted in the following illustration −   Process of Logon   * An intruder can only see the hashes of passwords, even if he accessed the password. He can neither logon using hash nor can he derive the password from hash value since hash function possesses the property of pre-image resistance.   Data Integrity Check  Data integrity check is a most common application of the hash functions. It is used to generate the checksums on data files. This application provides assurance to the user about correctness of the data.  The process is depicted in the following illustration −  Data Integrity Check  The integrity check helps the user to detect any changes made to original file. It however, does not provide any assurance about originality. The attacker, instead of modifying file data, can change the entire file and compute all together new hash and send to the receiver. This integrity check application is useful only if the user is sure about the originality of file. The MD5 cryptographic hash function The MD5 function is a cryptographic algorithm that takes an input of arbitrary length and produces a message digest that is 128 bits long. The digest is sometimes also called the "hash" or "fingerprint" of the input. MD5 is used in many situations where a potentially long message needs to be processed and/or compared quickly. The most common application is the creation and verification of [digital signatures](http://www.iusmentis.com/technology/digitalsignatures/).  MD5 was designed by well-known cryptographer Ronald Rivest in 1991. In 2004, some serious flaws were found in MD5. The complete implications of these flaws has yet to be determined. How MD5 worksPreparing the input The MD5 algorithm first divides the input in **blocks** of 512 bits each. 64 Bits are inserted at the end of the last block. These 64 bits are used to record the length of the original input. If the last block is less than 512 bits, some extra bits are 'padded' to the end.  Next, each **block** is divided into 16 **words** of 32 bits each. These are denoted as M0 ... M15. MD5 helper functionsThe buffer MD5 uses a buffer that is made up of four **words** that are each 32 bits long. These words are called A, B, C and D. They are initialized as  word A: 01 23 45 67  word B: 89 ab cd ef  word C: fe dc ba 98  word D: 76 54 32 10 The table MD5 further uses a table K that has 64 elements. Element number i is indicated as Ki. The table is computed beforehand to speed up the computations. The elements are computed using the mathematical sin function:  Ki = abs(sin(i + 1)) \* 232 Four auxiliary functions In addition MD5 uses four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word. They apply the logical operators and, or, not and xor to the input bits.  F(X,Y,Z) = (X and Y) or (not(X) and Z)  G(X,Y,Z) = (X and Z) or (Y and not(Z))  H(X,Y,Z) = X xor Y xor Z  I(X,Y,Z) = Y xor (X or not(Z)) Processing the blocks The contents of the four buffers (A, B, C and D) are now mixed with the words of the input, using the four auxiliary functions (F, G, H and I). There are four rounds, each involves 16 basic operations. One operation is illustrated in the figure below.  One operation performed in a round of the MD5 function  The figure shows how the auxiliary function F is applied to the four buffers (A, B, C and D), using message word Mi and constant Ki. The item "<<<s" denotes a binary left shift by s bits. The output After all rounds have been performed, the buffers A, B, C and D contain the MD5 digest of the original input. Knapsack Cryptosystem The **knapsack cryptosystem** is a [public-key cryptosystem](https://brilliant.org/wiki/public-key-cryptography/) based on a special case of the classic problem in [combinatorics](https://brilliant.org/wiki/learn-and-practice-combinatorics-on-brilliant/) known as the [knapsack problem](https://brilliant.org/wiki/backpack-problem/). It was developed by Ralph Merklee and Martin Hellman in 1978 and is one of the earliest public key cryptosystems. This special application of the knapsack problem is also akin to the [subset sum problem](https://brilliant.org/wiki/subset-sum/), where the solution is rather time consuming to compute as it belongs in the [complexity class](https://brilliant.org/wiki/complexity-classes/) of [NP](https://brilliant.org/wiki/p-versus-np/#NP-Algorithms). Key Generation The knapsack cryptosystem, also known as the Merklee-Hellman cipher, begins by generating user encryption keys as two knapsacks, one public and one private. The public key is the hard knapsack and the private key is the super-increasing knapsack. Two numbers, one called a **multiplier** and the other a **modulus**, are used to convert the super-increasing knapsack into the hard knapsack. These same numbers are used to convert the sum of the subsets of the hard knapsack into the sum of the subsets of the easy knapsack, which is a problem solvable in polynomial time. Algorithm A typical user first chooses a super-increasing sequence .Then a modulus and a multiplier such that are selected. This ensures that the congruence has a unique solution, say, .  Then the sequence of integers defined by  for all to where is selected. Carrying out this transformation generally destroys the super-increasing property enjoyed by .  The user keeps secret the original sequence and and , but publishes in a public directory. Anyone who wishes to send a message employs this publicly available sequence as the encryption key.  The sender begins by converting the plaintext message into a string of 's and 's by using the ASCII Alphabet binary representation of digits.  https://ds055uzetaobb.cloudfront.net/image_optimizer/7f187f4be15e9885cb517a882bb62309d794bcdc.jpg  The string is then split into blocks of binary digits. The public encrypting sequence is then used to transform the given plaintext, say, into the sum  Where the number S is the hidden information that the sender transmits over an insecure communication channel.  Since each is either or , the problem of obtaining the plaintext block from is equivalent to solving a difficult knapsack problem. However, the private key can be used to convert this hard knapsack into an easy one, allowing an efficient decryption proces.  Knowing the values of and , the recipient computes where  Since ,  Because was chosen such that , this leaves .Therefore, must hold. The solution of this super-increasing knapsack problem furnishes a solution to a difficult problem, and the plain-text block is recovered, which is converted back to its alphabetic representation.For a more thorough explanation of the knapsack cryptosystem, refer to [this worked example of the algorithm](https://brilliant.org/discussions/thread/the-knapsack-cryptosystem-at-work/). Security This cipher aroused a great deal of interest at the beginning because it was based on a probably hard problem, but in 1982 Adi Shamir provided a fast algorithm for solving knapsack problems. The main weakness of this system is that the sequence is too special. The system can be made difficult by changing the values of and at regular intervals (iterating) but even that variant was cracked in 1985. Since the knapsack cryptosystem is obsolete, the [RSA cipher](https://brilliant.org/discussions/thread/introduction-to-cryptographyrsa-cipher/) is now the dominant public key cryptography algorithm.   |  | | --- | | #define APP\_NAME "encryptor" | |  |  |  | | --- | |  | |  |  |  | | --- | | #include "ks\_crypt.h" | |  |  |  | | --- | | #include "uint1024.h" | |  |  |  | | --- | | #include "config.h" | |  |  |  | | --- | | #include <stdint.h> | |  |  |  | | --- | | #include <stdio.h> | |  |  |  | | --- | | #include <getopt.h> | |  |  |  | | --- | |  | |  |  |  | | --- | | char \*in\_fn, \*out\_fn, \*key\_fn="public-key"; | |  |  |  | | --- | | /\* files with key, input file & output file \*/ | |  |  |  | | --- | |  | |  |  |  | | --- | | void init(char \*pn); /\* prints stuff about prog&author \*/ | |  |  |  | | --- | | void warranty(void); /\* warranty - cut&pasted from GPL \*/ | |  |  |  | | --- | | void help(void); /\* prints help \*/ | |  |  |  | | --- | |  | |  |  |  | | --- | | int parse\_args(int argc, char \*argv[]) { | |  |  |  | | --- | | int c; | |  |  |  | | --- | | int opt\_ix; | |  |  |  | | --- | | static struct option long\_options[] = | |  |  |  | | --- | | { | |  |  |  | | --- | | { "warranty", 0, 0, 'w'}, | |  |  |  | | --- | | { "help", 0, 0, 0}, | |  |  |  | | --- | | { "key-file", 1, 0, 'k'}, | |  |  |  | | --- | | { 0, 0, 0, 0} | |  |  |  | | --- | | }; | |  |  |  | | --- | |  | |  |  |  | | --- | | while (1) { | |  |  |  | | --- | | c = getopt\_long (argc, argv, "wk:", | |  |  |  | | --- | | long\_options, &opt\_ix); | |  |  |  | | --- | |  | |  |  |  | | --- | | if (c==-1) break; | |  |  |  | | --- | |  | |  |  |  | | --- | | switch (c) { | |  |  |  | | --- | | case 0: | |  |  |  | | --- | | switch (opt\_ix) { | |  |  |  | | --- | | case 1: init(argv[0]); help(); return(1); | |  |  |  | | --- | | default: | |  |  |  | | --- | | return(1); | |  |  |  | | --- | | } | |  |  |  | | --- | | break; | |  |  |  | | --- | |  | |  |  |  | | --- | | case 'w': warranty(); return(1); | |  |  |  | | --- | |  | |  |  |  | | --- | | case 'v': | |  |  |  | | --- | | if (optarg) | |  |  |  | | --- | | if (sscanf(optarg,"%hd", &verbose)!=1) { | |  |  |  | | --- | | fprintf(stderr,"Invalid argument for %s: %s\n", | |  |  |  | | --- | | argv[optind], optarg); | |  |  |  | | --- | | return(1); | |  |  |  | | --- | | } else; | |  |  |  | | --- | | else verbose=1; | |  |  |  | | --- | | break; | |  |  |  | | --- | |  | |  |  |  | | --- | | case 'q': | |  |  |  | | --- | | verbose = -1; | |  |  |  | | --- | | break; | |  |  |  | | --- | |  | |  |  |  | | --- | | case 'k': key\_fn=optarg; break; | |  |  |  | | --- | |  | |  |  |  | | --- | | case '?': | |  |  |  | | --- | | return(1); | |  |  |  | | --- | | default: | |  |  |  | | --- | | fprintf(stderr, | |  |  |  | | --- | | "?? getopt returned character code 0%o ??\n", c); | |  |  |  | | --- | | } | |  |  |  | | --- | | } | |  |  |  | | --- | | if (optind<argc) out\_fn = argv[optind++]; else out\_fn = 0; | |  |  |  | | --- | |  | |  |  |  | | --- | | if (optind<argc) in\_fn = argv[optind++]; else in\_fn=0; | |  |  |  | | --- | |  | |  |  |  | | --- | | /\* init(argv[0]); \*/ | |  |  |  | | --- | |  | |  |  |  | | --- | | if (optind < argc) | |  |  |  | | --- | | { | |  |  |  | | --- | | fputs ("Unused parameter(s): ",stderr); | |  |  |  | | --- | | while (optind < argc) | |  |  |  | | --- | | fputs (argv[optind++],stderr); | |  |  |  | | --- | | fputs ("\n",stderr); | |  |  |  | | --- | | } | |  |  |  | | --- | | return(0); | |  |  |  | | --- | | } | |  |  |  | | --- | |  | |  |  |  | | --- | |  | |  |  |  | | --- | | /\*\*\*\*\*\*\*\*\*\*\*\* | |  |  |  | | --- | | \* MAIN \* | |  |  |  | | --- | | \*\*\*\*\*\*\*\*\*\*\*/ | |  |  |  | | --- | | int main(int argc, char \*argv[]) { | |  |  |  | | --- | |  | |  |  |  | | --- | | FILE \*fi,\*fo, \*ft; | |  |  |  | | --- | | uint8\_t data[ITEMS/8]; | |  |  |  | | --- | | uint1024 d; | |  |  |  | | --- | | int e; | |  |  |  | | --- | | uint8\_t r; | |  |  |  | | --- | |  | |  |  |  | | --- | | verbose = 1; | |  |  |  | | --- | | if (parse\_args(argc,argv)) return(1); | |  |  |  | | --- | |  | |  |  |  | | --- | | switch (load\_pub\_key(key\_fn)) { | |  |  |  | | --- | | case 0: break; | |  |  |  | | --- | | case -1: | |  |  |  | | --- | | fprintf(stderr,"Could not open %s.\n",key\_fn); | |  |  |  | | --- | | return(2); | |  |  |  | | --- | | default: | |  |  |  | | --- | | fprintf(stderr,"Incorrect format of public key.\n"); | |  |  |  | | --- | | return(3); | |  |  |  | | --- | | } | |  |  |  | | --- | |  | |  |  |  | | --- | | if (in\_fn && strcmp(in\_fn,"-")) | |  |  |  | | --- | | if (!(fi=fopen(in\_fn, "r"))) { | |  |  |  | | --- | | fprintf(stderr,"Could not open file %s.\n",in\_fn); | |  |  |  | | --- | | return(4); | |  |  |  | | --- | | } else; | |  |  |  | | --- | | else fi=stdin; | |  |  |  | | --- | |  | |  |  |  | | --- | | if (out\_fn && strcmp(out\_fn,"-")) | |  |  |  | | --- | | if (!(fo=fopen(out\_fn, "w"))) { | |  |  |  | | --- | | fprintf(stderr,"Could not create file %s.\n",out\_fn); | |  |  |  | | --- | | return(5); | |  |  |  | | --- | | } else; | |  |  |  | | --- | | else fo=stdout; | |  |  |  | | --- | |  | |  |  |  | | --- | | if (!(ft=tmpfile() )) { | |  |  |  | | --- | | fputs("Could not create temp file.\n",stderr); | |  |  |  | | --- | | return(6); | |  |  |  | | --- | | } | |  |  |  | | --- | |  | |  |  |  | | --- | | e=0; | |  |  |  | | --- | | while ( !feof(fi) && !ferror(fi) && !ferror(ft) ) { | |  |  |  | | --- | | r=fread(data, 1, ITEMS/8, fi); | |  |  |  | | --- | | if (r) { | |  |  |  | | --- | | for (e=r; e<ITEMS/8; e++) data[e]=0; | |  |  |  | | --- | | encrypt(data,d); | |  |  |  | | --- | | write1024(ft, d); | |  |  |  | | --- | | e=1; | |  |  |  | | --- | | } | |  |  |  | | --- | | } | |  |  |  | | --- | |  | |  |  |  | | --- | | if (ferror(fi)) { | |  |  |  | | --- | | fprintf(stderr, "Error encountered during reading from %s\n", | |  |  |  | | --- | | in\_fn); | |  |  |  | | --- | | return(7); | |  |  |  | | --- | | } | |  |  |  | | --- | | fclose(fi); | |  |  |  | | --- | |  | |  |  |  | | --- | | if (ferror(ft)) { | |  |  |  | | --- | | fprintf(stderr, "Could not write to temp file\n"); | |  |  |  | | --- | | return(8); | |  |  |  | | --- | | } | |  |  |  | | --- | |  | |  |  |  | | --- | | if (e && !r) r=ITEMS/8; | |  |  |  | | --- | | fwrite( &r, 1, 1, fo); | |  |  |  | | --- | | rewind(ft); | |  |  |  | | --- | | while (!ferror(ft) && !ferror(fo)) { | |  |  |  | | --- | | read1024(ft,d); | |  |  |  | | --- | | if (!feof(ft)) write1024(fo,d); else break; | |  |  |  | | --- | | } | |  |  |  | | --- | |  | |  |  |  | | --- | | if (ferror(ft)) { | |  |  |  | | --- | | fprintf(stderr, | |  |  |  | | --- | | "Error encountered during reading from temp file\n"); | |  |  |  | | --- | | return(7); | |  |  |  | | --- | | } | |  |  |  | | --- | | if (ferror(fo)) { | |  |  |  | | --- | | fprintf(stderr, "Could not write to %s\n", | |  |  |  | | --- | | (out\_fn)?out\_fn:"stdout"); | |  |  |  | | --- | | return(8); | |  |  |  | | --- | | } | |  |  |  | | --- | | fclose(fo); fclose(ft); | |  |  |  | | --- | |  | |  |  |  | | --- | | return(0); | |  |  |  | | --- | | } /\* main \*/ | |  |  |  | | --- | |  | |  |  |  | | --- | |  | |  |  |  | | --- | |  | |  |  |  | | --- | | /\*\*\*\* STUFF \*\*\*\*/ | |  |  |  | | --- | |  | |  |  |  | | --- | |  | |  |  |  | | --- | | void help(void) { | |  |  |  | | --- | | printf(" | |  |  |  | | --- | | Syntax: | |  |  |  | | --- | | %s [options] [output file] [input file] | |  |  |  | | --- | |  | |  |  |  | | --- | | If some file is not specified or is -, standard input/output will be used. | |  |  |  | | --- | |  | |  |  |  | | --- | | Options: | |  |  |  | | --- | | -w --warranty show warranty information | |  |  |  | | --- | | --help show this | |  |  |  | | --- | |  | |  |  |  | | --- | | -k --key-file specifies file containing public key | |  |  |  | | --- | | ",APP\_NAME); | |  |  |  | | --- | | } | |  |  |  | | --- | |  | |  |  |  | | --- | | void init(char \*pn) { | |  |  |  | | --- | | if (verbose<0) return; | |  |  |  | | --- | | printf("\ | |  |  |  | | --- | | Knapsack solving problem encryption - %s (version %s) | |  |  |  | | --- | | Copyright (C) 2001 Miroslav 'Mirco' Bajtos | |  |  |  | | --- | | %s comes with ABSOLUTELY NO WARRANTY; for details run | |  |  |  | | --- | | %s --warranty | |  |  |  | | --- | | This is free software, and you are welcome to redistribute it | |  |  |  | | --- | | under certain conditions; see GPL for details. | |  |  |  | | --- | |  | |  |  |  | | --- | | ",APP\_NAME,VERSION,APP\_NAME,pn); | |  |  |  | | --- | | } | |  |  |  | | --- | |  | |  |  |  | | --- | | void warranty(void) { | |  |  |  | | --- | | puts("\ | | Mislanius damage issue |  |  | | --- | |  |  |  | | --- | | "); | |  |  |  | | --- | | } | |  | |