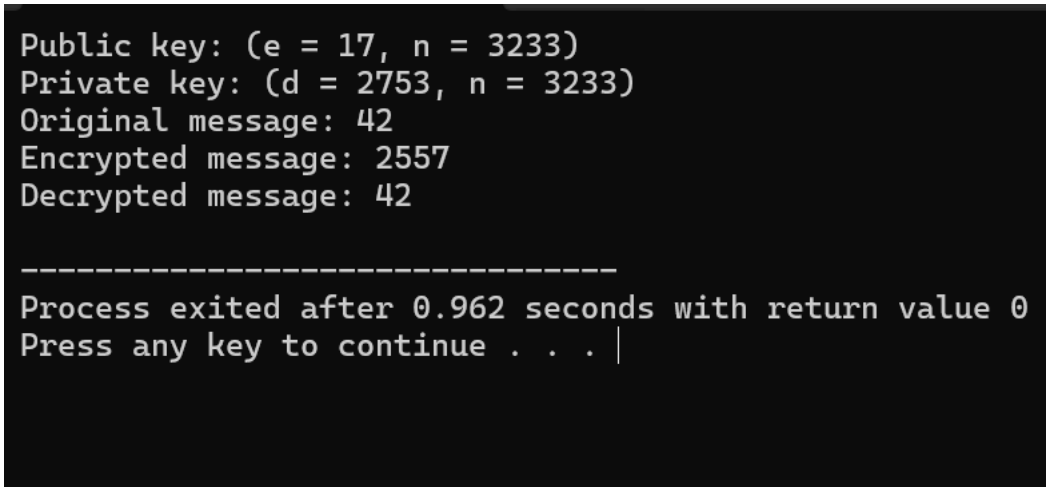
26.Code:

```
#include <stdio.h>
#include <stdlib.h>

long gcd(long a, long b) {
    while (b != 0) {
        long t = b;
        b = a % b;
        a = t;
    }
    return a;
}

long modInverse(long e, long phi) {
    long t = 0, newt = 1;
    long r = phi, newr = e;
    while (newr != 0) {
        long quotient = r / newr;
        long temp = newt;
        newt = t - quotient * newt;
        t = temp;

        temp = newr;
        newr = r - quotient * newr;
        r = temp;
    }
    if (r > 1) return -1;  // No inverse
    if (t < 0) t += phi;
    return t;
```

```
}

long modExp(long base, long exp, long mod) {
    long result = 1;
    base %= mod;
    while (exp > 0) {
        if (exp % 2 == 1) result = (result * base) % mod;
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}

int main() {
    long p = 61, q = 53; // small primes
    long n = p * q;
    long phi = (p - 1) * (q - 1);
    long e = 17; // public exponent

    if (gcd(e, phi) != 1) {
        printf("e and phi(n) are not coprime.\n");
        return 1;
    }

    long d = modInverse(e, phi); // private key
    printf("Public key: (e = %ld, n = %ld)\n", e, n);
    printf("Private key: (d = %ld, n = %ld)\n", d, n);

    long message = 42;
```

```
    long ciphertext = modExp(message, e, n);
    long decrypted = modExp(ciphertext, d, n);

    printf("Original message: %ld\n", message);
    printf("Encrypted message: %ld\n", ciphertext);
    printf("Decrypted message: %ld\n", decrypted);
    return 0;
}
```

Output:



27.Code:

```
#include <stdio.h>
#include <stdlib.h>
long modExp(long base, long exp, long mod) {
    long result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1) result = (result * base) % mod;
        exp = exp >> 1;
        base = (base * base) % mod;
    }
```

```
    return result;
}
int charToInt(char c) {
    if (c >= 'A' && c <= 'Z') return c - 'A';
    if (c >= 'a' && c <= 'z') return c - 'a';
    return -1;
}
char intToChar(int i) {
    return 'A' + (i % 26);
}
int main() {
    long e = 17;
    long n = 3233;
    char message[] = "HELLO";
    printf("Original: %s\nEncrypted: ", message);
    for (int i = 0; message[i] != '\0'; i++) {
        int m = charToInt(message[i]);
        if (m >= 0) {
            long c = modExp(m, e, n);
            printf("%ld ", c);
        }
    }
    printf("\n");
    return 0;
}
```

Output:

```
Original: HELLO
Encrypted: 2369 1387 3061 3061 2549

--------------------------------
Process exited after 0.8935 seconds with return value 0
Press any key to continue . . .
```

28.Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
long modExp(long base, long exp, long mod) {
    long result = 1;
    base %= mod;
    while (exp > 0) {
        if (exp % 2 == 1) result = (result * base) % mod;
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}
int main() {
    long q = 353;
    long a = 3;
    long xA = 97;
    long xB = 233;
    long yA = modExp(a, xA, q);
    long yB = modExp(a, xB, q);
    long kA = modExp(yB, xA, q);
    long kB = modExp(yA, xB, q);
```

```
    printf("Alice's public value: %ld\n", yA);
    printf("Bob's public value: %ld\n", yB);
    printf("Shared secret (Alice): %ld\n", kA);
    printf("Shared secret (Bob): %ld\n", kB);
    return 0;
}
```

Output:

```
Alice's public value: 40
Bob's public value: 248
Shared secret (Alice): 160
Shared secret (Bob): 160

--------------------------------
Process exited after 0.9068 seconds with return value 0
Press any key to continue . . .
```

29.Code:

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>
#define TOTAL_LANES 25
#define RATE_LANES 16
#define CAPACITY_LANES 9
uint64_t random_nonzero() {
    uint64_t val = 0;
    while (val == 0)
        val = ((uint64_t)rand() << 32) | rand();
    return val;
}
```

```
int all_nonzero(uint64_t *lanes, int start, int count) {
    for (int i = start; i < start + count; i++) {
        if (lanes[i] == 0)
            return 0;
    }
    return 1;
}
int main() {
    srand(time(NULL));
    uint64_t state[TOTAL_LANES] = {0};
    int rounds = 0;
    for (int i = 0; i < RATE_LANES; i++) {
        state[i] = random_nonzero();
    }
    while (!all_nonzero(state, RATE_LANES, CAPACITY_LANES)) {
        rounds++;
        for (int i = 0; i < RATE_LANES; i++) {
            state[i] ^= random_nonzero();
        }
        for (int i = RATE_LANES; i < TOTAL_LANES; i++) {
            int r = rand() % RATE_LANES;
            state[i] ^= state[r];
        }
    }
    printf("All capacity lanes became nonzero after %d block(s).\n", rounds + 1);
    return 0;
}
```

Output:

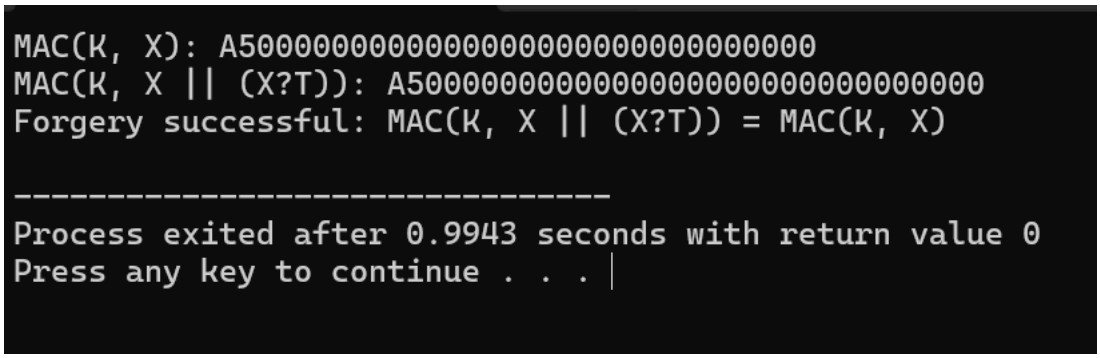
```
All capacity lanes became nonzero after 2 block(s).

--------------------------------
Process exited after 0.9579 seconds with return value 0
Press any key to continue . . .
```

30.Code:

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#define BLOCK_SIZE 16
void xor_block(uint8_t *out, const uint8_t *a, const uint8_t *b) {
    for (int i = 0; i < BLOCK_SIZE; i++)
        out[i] = a[i] ^ b[i];
}
void encrypt_block(uint8_t *out, const uint8_t *in, const uint8_t *key) {
    for (int i = 0; i < BLOCK_SIZE; i++)
        out[i] = in[i] ^ key[i];
}
void cbc_mac(uint8_t *mac, const uint8_t *msg, int blocks, const uint8_t *key) {
    uint8_t buffer[BLOCK_SIZE] = {0};
    for (int i = 0; i < blocks; i++) {
        xor_block(buffer, buffer, msg + i * BLOCK_SIZE);
        encrypt_block(buffer, buffer, key);
    }
    memcpy(mac, buffer, BLOCK_SIZE);
}
void print_block(const char *label, const uint8_t *block) {
    printf("%s: ", label);
    for (int i = 0; i < BLOCK_SIZE; i++)
```

```
        printf("%02X", block[i]);
    printf("\n");
}
int main() {
    uint8_t key[BLOCK_SIZE] = {0x0F};
    uint8_t X[BLOCK_SIZE]   = {0xAA};
    uint8_t T[BLOCK_SIZE];
    cbc_mac(T, X, 1, key);
    print_block("MAC(K, X)", T);
    uint8_t forged[BLOCK_SIZE * 2];
    memcpy(forged, X, BLOCK_SIZE);
    xor_block(forged + BLOCK_SIZE, X, T);
    uint8_t T_forged[BLOCK_SIZE];
    cbc_mac(T_forged, forged, 2, key);
    print_block("MAC(K, X || (X?T))", T_forged);
    if (memcmp(T, T_forged, BLOCK_SIZE) == 0)
        printf("Forgery successful: MAC(K, X || (X?T)) = MAC(K, X)\n");
    else
        printf("Forgery failed.\n");
    return 0;
}
```

Output:



31.Code:

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#define BLOCK_SIZE_128 16
#define BLOCK_SIZE_64 8
const uint8_t Rb_128[16] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                            0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x87};
const uint8_t Rb_64[8]   = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1B};
void mock_cipher(const uint8_t *in, uint8_t *out, int block_size) {
    memcpy(out, in, block_size);
}
void left_shift(uint8_t *out, const uint8_t *in, int block_size) {
    uint8_t overflow = 0;
    for (int i = block_size - 1; i >= 0; i--) {
        out[i] = (in[i] << 1) | overflow;
        overflow = (in[i] & 0x80) ? 1 : 0;
    }
}
void xor_block(uint8_t *out, const uint8_t *a, const uint8_t *b, int size) {
    for (int i = 0; i < size; i++)
        out[i] = a[i] ^ b[i];
}
void print_block(const char *label, const uint8_t *block, int size) {
    printf("%s: ", label);
    for (int i = 0; i < size; i++)
        printf("%02X", block[i]);
    printf("\n");
}
```

```
void generate_subkeys(int block_size) {
    uint8_t L[16] = {0};
    uint8_t K1[16] = {0};
    uint8_t K2[16] = {0};
    const uint8_t *Rb = (block_size == 16) ? Rb_128 : Rb_64;
    uint8_t zero[16] = {0};
    mock_cipher(zero, L, block_size);
    left_shift(K1, L, block_size);
    if (L[0] & 0x80) xor_block(K1, K1, Rb, block_size);
    left_shift(K2, K1, block_size);
    if (K1[0] & 0x80) xor_block(K2, K2, Rb, block_size);
    printf("\nBlock size: %d bits\n", block_size * 8);
    print_block("L", L, block_size);
    print_block("K1", K1, block_size);
    print_block("K2", K2, block_size);
}
int main() {
    generate_subkeys(BLOCK_SIZE_64);
    generate_subkeys(BLOCK_SIZE_128);
    return 0;
}
```

Output:

```
Block size: 64 bits
L: 0000000000000000
K1: 0000000000000000
K2: 0000000000000000

Block size: 128 bits
L: 00000000000000000000000000000000
K1: 00000000000000000000000000000000
K2: 00000000000000000000000000000000

--------------------------------
Process exited after 0.9017 seconds with return value 0
Press any key to continue . . .
```

32.Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
uint64_t mod_exp(uint64_t base, uint64_t exp, uint64_t mod) {
    uint64_t result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2)
            result = (result * base) % mod;
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}
int64_t modinv(int64_t a, int64_t m) {
    int64_t m0 = m, t, q;
```

```
    int64_t x0 = 0, x1 = 1;
    if (m == 1) return 0;
    while (a > 1) {
        q = a / m;
        t = m;
        m = a % m, a = t;
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
    }
    return (x1 < 0) ? x1 + m0 : x1;
}
uint64_t simple_hash(const char *msg) {
    uint64_t hash = 0;
    for (int i = 0; msg[i]; i++)
        hash = (hash * 31 + msg[i]) % 1000003;
    return hash;
}
void dsa_sign(const char *msg, uint64_t p, uint64_t q, uint64_t g, uint64_t x, uint64_t *r_out,
uint64_t *s_out) {
    uint64_t k, r, s;
    uint64_t h = simple_hash(msg);
    do {
        k = rand() % q;
    } while (k == 0);
    r = mod_exp(g, k, p) % q;
    s = (modinv(k, q) * (h + x * r)) % q;
    *r_out = r;
    *s_out = s;
```

```
}
uint64_t rsa_sign(const char *msg, uint64_t d, uint64_t n) {
    uint64_t h = simple_hash(msg);
    return mod_exp(h, d, n);
}
int main() {
    srand(time(NULL));
    const char *message = "Hello DSA vs RSA";
    uint64_t p = 467, q = 233, g = 2;
    uint64_t x = 123;
    uint64_t r1, s1, r2, s2;
    dsa_sign(message, p, q, g, x, &r1, &s1);
    dsa_sign(message, p, q, g, x, &r2, &s2);
    printf("DSA Signature 1: (r=%llu, s=%llu)\n", r1, s1);
    printf("DSA Signature 2: (r=%llu, s=%llu)\n", r2, s2);
    printf("%s\n", (r1 != r2 || s1 != s2) ? "? DSA signatures are different." : "? DSA signatures
are same (BAD)");
    uint64_t n = 391, d = 61;
    uint64_t sig1 = rsa_sign(message, d, n);
    uint64_t sig2 = rsa_sign(message, d, n);
    printf("\nRSA Signature 1: %llu\n", sig1);
    printf("RSA Signature 2: %llu\n", sig2);
    printf("%s\n", (sig1 == sig2) ? "? RSA signatures are same." : "? RSA signatures are different
(BAD)");
    return 0;
}
```

Output:

```
DSA Signature 1: (r=4, s=156)
DSA Signature 2: (r=63, s=24)
? DSA signatures are different.

RSA Signature 1: 85
RSA Signature 2: 85
? RSA signatures are same.

--------------------------------
Process exited after 0.5687 seconds with return value 0
Press any key to continue . . .
```

33.Code:

```
#include <stdio.h>

#include <string.h>

#include <stdint.h>

#define ROUNDS 16

int IP[] = {

    58,50,42,34,26,18,10,2,

    60,52,44,36,28,20,12,4,

    62,54,46,38,30,22,14,6,

    64,56,48,40,32,24,16,8,

    57,49,41,33,25,17,9,1,

    59,51,43,35,27,19,11,3,

    61,53,45,37,29,21,13,5,

    63,55,47,39,31,23,15,7

};

int FP[] = {

    40,8,48,16,56,24,64,32,

    39,7,47,15,55,23,63,31,

    38,6,46,14,54,22,62,30,

    37,5,45,13,53,21,61,29,

    36,4,44,12,52,20,60,28,
```

```
    35,3,43,11,51,19,59,27,
    34,2,42,10,50,18,58,26,
    33,1,41,9,49,17,57,25
};
uint32_t feistel(uint32_t half, uint64_t subkey) {
    return (half ^ (subkey & 0xFFFFFFFF));
}
void permute(uint8_t *in, uint8_t *out, int *table) {
    for (int i = 0; i < 64; i++) {
        int bit = (in[(table[i] - 1) / 8] >> (7 - ((table[i] - 1) % 8))) & 1;
        out[i / 8] |= bit << (7 - (i % 8));
    }
}
void des_encrypt_block(uint8_t *in, uint8_t *out, uint64_t key, int encrypt) {
    uint8_t permuted[8] = {0};
    permute(in, permuted, IP);
    uint32_t L = (permuted[0] << 24) | (permuted[1] << 16) | (permuted[2] << 8) | permuted[3];
    uint32_t R = (permuted[4] << 24) | (permuted[5] << 16) | (permuted[6] << 8) | permuted[7];
    for (int i = 0; i < ROUNDS; i++) {
        int round = encrypt ? i : (ROUNDS - 1 - i);
        uint64_t subkey = (key >> (round % 56)) & 0xFFFFFFFFFFFF;
        uint32_t temp = R;
        R = L ^ feistel(R, subkey);
        L = temp;
    }
    uint8_t preoutput[8] = {
        (R >> 24) & 0xFF, (R >> 16) & 0xFF, (R >> 8) & 0xFF, R & 0xFF,
        (L >> 24) & 0xFF, (L >> 16) & 0xFF, (L >> 8) & 0xFF, L & 0xFF
    };
```

```
    memset(out, 0, 8);
    permute(preoutput, out, FP);
}
void print_block(const char *label, uint8_t *block) {
    printf("%s: ", label);
    for (int i = 0; i < 8; i++)
        printf("%02X ", block[i]);
    printf("\n");
}
int main() {
    uint8_t plaintext[8] = "OpenAI!";
    uint8_t ciphertext[8] = {0};
    uint8_t decrypted[8] = {0};
    uint64_t key = 0x133457799BBCDFF1;
    printf("DES Demo\n---------\n");
    print_block("Plaintext", plaintext);
    des_encrypt_block(plaintext, ciphertext, key, 1);
    print_block("Encrypted", ciphertext);
    des_encrypt_block(ciphertext, decrypted, key, 0);
    print_block("Decrypted", decrypted);
    return 0;
}
```

Output:

```
DES Demo
---------
Plaintext: 4F 70 65 6E 41 49 21 00
Encrypted: C5 B1 4E 20 5E DD 63 AE
Decrypted: 4F 70 65 6E 41 49 21 00

--------------------------------
Process exited after 0.8495 seconds with return value 0
Press any key to continue . . .
```

34.Code:

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#define BLOCK_SIZE 8
#define MAX_LEN 128
void pad(uint8_t *input, int len, uint8_t *output, int *padded_len) {
    int pad_len = BLOCK_SIZE - (len % BLOCK_SIZE);
    memcpy(output, input, len);
    output[len] = 0x80;
    for (int i = 1; i < pad_len; i++)
        output[len + i] = 0x00;
    *padded_len = len + pad_len;
}
void xor_block(uint8_t *out, uint8_t *in1, uint8_t *in2) {
    for (int i = 0; i < BLOCK_SIZE; i++)
        out[i] = in1[i] ^ in2[i];
}
void print_hex(const char *label, uint8_t *data, int len) {
    printf("%s: ", label);
    for (int i = 0; i < len; i++)
        printf("%02X ", data[i]);
```

```
    printf("\n");
}
void ecb_encrypt(uint8_t *plaintext, uint8_t *ciphertext, int len, uint8_t *key) {
    for (int i = 0; i < len; i += BLOCK_SIZE)
        xor_block(ciphertext + i, plaintext + i, key);
}
void cbc_encrypt(uint8_t *plaintext, uint8_t *ciphertext, int len, uint8_t *key, uint8_t *iv) {
    uint8_t prev[BLOCK_SIZE];
    memcpy(prev, iv, BLOCK_SIZE);
    for (int i = 0; i < len; i += BLOCK_SIZE) {
        uint8_t xored[BLOCK_SIZE];
        xor_block(xored, plaintext + i, prev);
        xor_block(ciphertext + i, xored, key);
        memcpy(prev, ciphertext + i, BLOCK_SIZE);
    }
}
void cfb_encrypt(uint8_t *plaintext, uint8_t *ciphertext, int len, uint8_t *key, uint8_t *iv) {
    uint8_t shift[BLOCK_SIZE];
    memcpy(shift, iv, BLOCK_SIZE);
    for (int i = 0; i < len; i++) {
        uint8_t encrypted[BLOCK_SIZE];
        xor_block(encrypted, shift, key);
        ciphertext[i] = plaintext[i] ^ encrypted[0];
        memmove(shift, shift + 1, BLOCK_SIZE - 1);
        shift[BLOCK_SIZE - 1] = ciphertext[i];
    }
}
int main() {
    uint8_t key[BLOCK_SIZE] = {0x0F, 0x1E, 0x2D, 0x3C, 0x4B, 0x5A, 0x69, 0x78};
```

```
    uint8_t iv[BLOCK_SIZE]  = {0xAA, 0xBB, 0xCC, 0xDD, 0x11, 0x22, 0x33, 0x44};
    uint8_t plaintext[MAX_LEN] = "HELLO BLOCK CIPHER";
    uint8_t padded[MAX_LEN] = {0}, ct_ecb[MAX_LEN] = {0}, ct_cbc[MAX_LEN] = {0},
ct_cfb[MAX_LEN] = {0};
    int padded_len;
    pad(plaintext, strlen((char *)plaintext), padded, &padded_len);
    ecb_encrypt(padded, ct_ecb, padded_len, key);
    cbc_encrypt(padded, ct_cbc, padded_len, key, iv);
    cfb_encrypt(padded, ct_cfb, padded_len, key, iv);
    printf("Plaintext: %s\n", plaintext);
    print_hex("ECB Ciphertext", ct_ecb, padded_len);
    print_hex("CBC Ciphertext", ct_cbc, padded_len);
    print_hex("CFB Ciphertext", ct_cfb, padded_len);
    return 0;
}
```

Output:

```
Plaintext: HELLO BLOCK CIPHER
ECB Ciphertext: 47 5B 61 70 04 7A 2B 34 40 5D 66 1C 08 13 39 30 4A 4C AD 3C 4B 5A 69 78
CBC Ciphertext: ED E0 AD AD 15 58 18 70 AD BD CB B1 1D 4B 21 40 E7 F1 66 8D 56 11 48 38
CFB Ciphertext: ED F1 8F 9E 51 0D 7E 07 AD BD CB B1 1D 4B 21 40 E7 E0 44 BE 12 44 2E 4F

--------------------------------
Process exited after 1.212 seconds with return value 0
Press any key to continue . . .
```

35.Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#define MAX_LEN 100
char encrypt_char(char p, int k) {
    if (p >= 'A' && p <= 'Z')
        return ((p - 'A' + k) % 26) + 'A';
```

```
    return p;
}
char decrypt_char(char c, int k) {
    if (c >= 'A' && c <= 'Z')
        return ((c - 'A' - k + 26) % 26) + 'A';
    return c;
}
int main() {
    char plaintext[MAX_LEN], ciphertext[MAX_LEN], decrypted[MAX_LEN];
    int key[MAX_LEN];
    int len;
    printf("Enter UPPERCASE plaintext (no spaces, max 100 letters): ");
    scanf("%s", plaintext);
    len = strlen(plaintext);
    srand(time(NULL));
    printf("Random Key: ");
    for (int i = 0; i < len; i++) {
        key[i] = rand() % 27;
        printf("%d ", key[i]);
    }
    printf("\n");
    for (int i = 0; i < len; i++) {
        ciphertext[i] = encrypt_char(plaintext[i], key[i]);
    }
    ciphertext[len] = '\0';
    printf("Ciphertext: %s\n", ciphertext);
    for (int i = 0; i < len; i++) {
        decrypted[i] = decrypt_char(ciphertext[i], key[i]);
    }
```

```
    decrypted[len] = '\0';
    printf("Decrypted : %s\n", decrypted);
    return 0;
}
```

Output:

```
Enter UPPERCASE plaintext (no spaces, max 100 letters): SUBHASHREDDY
Random Key: 18 4 0 25 17 2 9 14 24 15 6 18
Ciphertext: KYBGRUQFCSJQ
Decrypted : SUBHASHREDDY

--------------------------------
Process exited after 17.03 seconds with return value 0
Press any key to continue . . .
```

36.Code:

```
#include <stdio.h>
#include <string.h>
int gcd(int a, int b) {
    while (b) {
        int t = a % b;
        a = b;
        b = t;
    }
    return a;
}
int mod_inverse(int a, int m) {
    for (int i = 1; i < m; i++) {
        if ((a * i) % m == 1)
            return i;
    }
    return -1;
}
char encrypt_char(char p, int a, int b) {
```

```
    if (p >= 'A' && p <= 'Z') {
        int x = p - 'A';
        int c = (a * x + b) % 26;
        return c + 'A';
    }
    return p;
}
char decrypt_char(char c, int a, int b) {
    if (c >= 'A' && c <= 'Z') {
        int a_inv = mod_inverse(a, 26);
        if (a_inv == -1) return '?';
        int y = c - 'A';
        int p = (a_inv * (y - b + 26)) % 26;
        return p + 'A';
    }
    return c;
}
int main() {
    char plaintext[100], ciphertext[100], decrypted[100];
    int a, b;
    printf("Enter plaintext in UPPERCASE (no spaces): ");
    scanf("%s", plaintext);
    printf("Enter keys a and b (a must be coprime with 26): ");
    scanf("%d %d", &a, &b);
    if (gcd(a, 26) != 1) {
        printf("ERROR: 'a' must be coprime with 26 for invertibility.\n");
        return 1;
    }
    int len = strlen(plaintext);
```

```
    for (int i = 0; i < len; i++)
        ciphertext[i] = encrypt_char(plaintext[i], a, b);
    ciphertext[len] = '\0';
    for (int i = 0; i < len; i++)
        decrypted[i] = decrypt_char(ciphertext[i], a, b);
    decrypted[len] = '\0';
    printf("Ciphertext: %s\n", ciphertext);
    printf("Decrypted : %s\n", decrypted);
    return 0;
}
```

Output:

```
Enter plaintext in UPPERCASE (no spaces): SUBHASHREDDY
Enter keys a and b (a must be coprime with 26): b
ERROR: 'a' must be coprime with 26 for invertibility.

--------------------------------
Process exited after 12.31 seconds with return value 1
Press any key to continue . . .
```

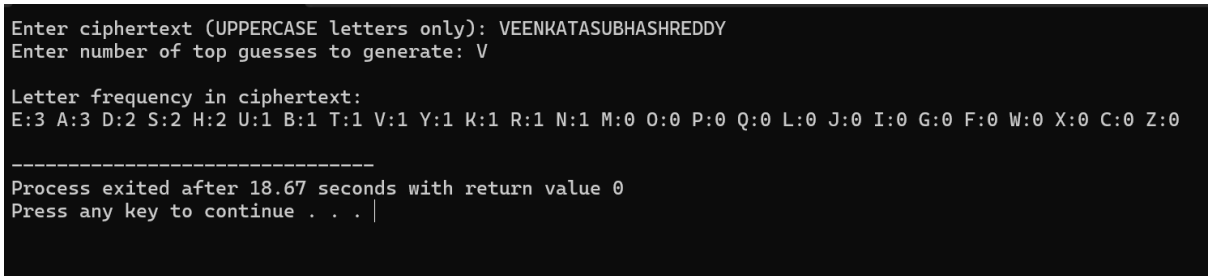
37.Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX_TEXT 1000
#define ALPHABET 26
const char *english_freq = "ETAOINSHRDLCUMWFGYPBVKJXQZ";
typedef struct {
    char letter;
    int freq;
} LetterFreq;
int cmp(const void *a, const void *b) {
```

```
    return ((LetterFreq *)b)->freq - ((LetterFreq *)a)->freq;
}
void apply_mapping(char *cipher, char *map, char *result) {
    for (int i = 0; cipher[i]; i++) {
        if (isupper(cipher[i]))
            result[i] = map[cipher[i] - 'A'];
        else
            result[i] = cipher[i];
    }
    result[strlen(cipher)] = '\0';
}
int main() {
    char ciphertext[MAX_TEXT];
    LetterFreq freq[ALPHABET];
    char mapping[ALPHABET], plaintext[MAX_TEXT];
    int topN;
    printf("Enter ciphertext (UPPERCASE letters only): ");
    scanf("%s", ciphertext);
    printf("Enter number of top guesses to generate: ");
    scanf("%d", &topN);
    for (int i = 0; i < ALPHABET; i++) {
        freq[i].letter = 'A' + i;
        freq[i].freq = 0;
    }
    for (int i = 0; ciphertext[i]; i++) {
        if (isupper(ciphertext[i]))
            freq[ciphertext[i] - 'A'].freq++;
    }
    qsort(freq, ALPHABET, sizeof(LetterFreq), cmp);
```

```
    printf("\nLetter frequency in ciphertext:\n");
    for (int i = 0; i < ALPHABET; i++)
        printf("%c:%d ", freq[i].letter, freq[i].freq);
    printf("\n");
    for (int guess = 0; guess < topN && guess < ALPHABET; guess++) {
        for (int i = 0; i < ALPHABET; i++)
            mapping[freq[i].letter - 'A'] = english_freq[(i + guess) % ALPHABET];
        apply_mapping(ciphertext, mapping, plaintext);
        printf("Guess #%d: %s\n", guess + 1, plaintext);
    }
    return 0;
}
```

Output:



38.Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define SIZE 2
#define MOD 26
int modInverse(int a, int m) {
    a = a % m;
    for (int x = 1; x < m; x++)
        if ((a * x) % m == 1)
            return x;
```

```
    return -1;
}
int determinant(int key[SIZE][SIZE]) {
    return (key[0][0] * key[1][1] - key[0][1] * key[1][0]) % MOD;
}
int invertKey(int key[SIZE][SIZE], int invKey[SIZE][SIZE]) {
    int det = determinant(key);
    int detInv = modInverse((det + MOD) % MOD, MOD);
    if (detInv == -1)
        return 0;
    invKey[0][0] = ( key[1][1] * detInv) % MOD;
    invKey[0][1] = (-key[0][1] * detInv + MOD) % MOD;
    invKey[1][0] = (-key[1][0] * detInv + MOD) % MOD;
    invKey[1][1] = ( key[0][0] * detInv) % MOD;
    return 1;
}
void encryptBlock(int key[SIZE][SIZE], int pt[SIZE], int ct[SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        ct[i] = 0;
        for (int j = 0; j < SIZE; j++)
            ct[i] += key[i][j] * pt[j];
        ct[i] %= MOD;
    }
}
void textToVec(char *text, int vec[SIZE]) {
    for (int i = 0; i < SIZE; i++)
        vec[i] = text[i] - 'A';
}
void printVec(int vec[SIZE]) {
```

```
    for (int i = 0; i < SIZE; i++)
        printf("%c", vec[i] + 'A');
}
void recoverKey(int pt[SIZE][SIZE], int ct[SIZE][SIZE], int key[SIZE][SIZE]) {
    int ptInv[SIZE][SIZE];
    if (!invertKey(pt, ptInv)) {
        printf("Plaintext matrix is not invertible!\n");
        return;
    }
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            key[i][j] = 0;
            for (int k = 0; k < SIZE; k++) {
                key[i][j] += ct[i][k] * ptInv[k][j];
            }
            key[i][j] %= MOD;
        }
    }
}
int main() {
    int key[SIZE][SIZE] = {{3, 3}, {2, 5}};
    char plaintext[] = "HI";
    int ptVec[SIZE], ctVec[SIZE];
    printf("Original Key Matrix:\n");
    for (int i = 0; i < SIZE; i++)
        printf("%d %d\n", key[i][0], key[i][1]);
    textToVec(plaintext, ptVec);
    encryptBlock(key, ptVec, ctVec);
    printf("\nPlaintext: %s", plaintext);
```

```
    printf("\nCiphertext: ");
    printVec(ctVec);
    printf("\n");
    int ptMat[SIZE][SIZE] = {{'H' - 'A', 'I' - 'A'}, {'H' - 'A', 'I' - 'A'}};
    int ctMat[SIZE][SIZE] = {{ctVec[0], ctVec[1]}, {ctVec[0], ctVec[1]}};
    int recoveredKey[SIZE][SIZE];
    recoverKey(ptMat, ctMat, recoveredKey);
    printf("\nRecovered Key Matrix:\n");
    for (int i = 0; i < SIZE; i++)
        printf("%d %d\n", (recoveredKey[i][0] + MOD) % MOD, (recoveredKey[i][1] + MOD)
% MOD);
    return 0;
}
```

Output:

```
Original Key Matrix:
3 3
2 5

Plaintext: HI
Ciphertext: TC
Plaintext matrix is not invertible!

Recovered Key Matrix:
0 0
0 0

--------------------------------
Process exited after 1.214 seconds with return value 0
Press any key to continue . . .
```

39.Code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX_TEXT 1000
```

```
#define ALPHABET 26
void decrypt(char *cipher, int key, char *output) {
    for (int i = 0; cipher[i]; i++) {
        if (isupper(cipher[i])) {
            output[i] = (cipher[i] - 'A' - key + ALPHABET) % ALPHABET + 'A';
        } else {
            output[i] = cipher[i];
        }
    }
    output[strlen(cipher)] = '\0';
}
char get_most_frequent_letter(char *text) {
    int freq[ALPHABET] = {0};
    for (int i = 0; text[i]; i++) {
        if (isupper(text[i])) {
            freq[text[i] - 'A']++;
        }
    }
    int max = 0, index = 0;
    for (int i = 0; i < ALPHABET; i++) {
        if (freq[i] > max) {
            max = freq[i];
            index = i;
        }
    }
    return 'A' + index;
}
int main() {
    char ciphertext[MAX_TEXT], plaintext[MAX_TEXT];
```

```
    int topN;
    printf("Enter ciphertext (UPPERCASE letters only): ");
    scanf("%s", ciphertext);
    printf("Enter number of top guesses to generate: ");
    scanf("%d", &topN);
    char most_freq = get_most_frequent_letter(ciphertext);
    int probable_key = (most_freq - 'E' + ALPHABET) % ALPHABET;
    printf("\nMost frequent letter: %c\n", most_freq);
    printf("Likely key guess based on 'E': %d\n", probable_key);
    printf("\nTop %d probable plaintexts:\n", topN);
    for (int i = 0; i < topN && i < ALPHABET; i++) {
        int key = (probable_key + i) % ALPHABET;
        decrypt(ciphertext, key, plaintext);
        printf("Key %2d: %s\n", key, plaintext);
    }
    return 0;
}
```

Output:

```
Enter ciphertext (UPPERCASE letters only): SANEKOMMU
Enter number of top guesses to generate: s

Most frequent letter: M
Likely key guess based on 'E': 8

Top 0 probable plaintexts:

--------------------------------
Process exited after 7.601 seconds with return value 0
Press any key to continue . . .
```

40.Code:

```
#include <stdio.h>
#include <string.h>
```

```
#include <ctype.h>
#include <stdlib.h>
#define MAX_TEXT 1000
#define ALPHABET 26
#define MAX_GUESSES 26
const char ENGLISH_FREQ[] = "ETAOINSHRDLCUMWFGYPBVKJXQZ";
void count_frequency(const char *text, int freq[ALPHABET]) {
    for (int i = 0; i < ALPHABET; i++) freq[i] = 0;
    for (int i = 0; text[i]; i++) {
        if (isupper(text[i])) {
            freq[text[i] - 'A']++;
        }
    }
}
void sort_by_frequency(int freq[ALPHABET], char *map) {
    int sorted[ALPHABET];
    for (int i = 0; i < ALPHABET; i++) sorted[i] = i;
    for (int i = 0; i < ALPHABET - 1; i++) {
        for (int j = i + 1; j < ALPHABET; j++) {
            if (freq[sorted[j]] > freq[sorted[i]]) {
                int tmp = sorted[i];
                sorted[i] = sorted[j];
                sorted[j] = tmp;
            }
        }
    }
    for (int i = 0; i < ALPHABET; i++) {
        map[i] = 'A' + sorted[i];
    }
```

```
}
void substitute(const char *cipher, const char *cipher_map, const char *eng_map, char *output) {
    for (int i = 0; cipher[i]; i++) {
        if (isupper(cipher[i])) {
            char *p = strchr(cipher_map, cipher[i]);
            if (p) {
                int idx = p - cipher_map;
                output[i] = eng_map[idx];
            } else {
                output[i] = cipher[i];
            }
        } else {
            output[i] = cipher[i];
        }
    }
    output[strlen(cipher)] = '\0';
}
int main() {
    char cipher[MAX_TEXT], cipher_map[ALPHABET + 1];
    int freq[ALPHABET], n;
    printf("Enter ciphertext (UPPERCASE letters only): ");
    scanf("%s", cipher);
    printf("Enter number of top guesses to generate (max %d): ", MAX_GUESSES);
    scanf("%d", &n);
    if (n > MAX_GUESSES) n = MAX_GUESSES;
    count_frequency(cipher, freq);
    sort_by_frequency(freq, cipher_map);
    cipher_map[ALPHABET] = '\0';
```

```
    printf("\nTop %d guesses using frequency analysis:\n", n);
    for (int i = 0; i < n; i++) {
        char guess[MAX_TEXT], eng_map[ALPHABET + 1];
        for (int j = 0; j < ALPHABET; j++)
            eng_map[j] = ENGLISH_FREQ[(j + i) % ALPHABET];
        eng_map[ALPHABET] = '\0';
        substitute(cipher, cipher_map, eng_map, guess);
        printf("Guess %2d: %s\n", i + 1, guess);
    }
    return 0;
}
```

Output:

```
Enter ciphertext (UPPERCASE letters only): SUBHASH
Enter number of top guesses to generate (max 26): e

Top 0 guesses using frequency analysis:

--------------------------------
Process exited after 11.81 seconds with return value 0
Press any key to continue . . .
```