Codificación

Autor: Félix Moreno Peñarrubia

Solución en $3n + 2$ bits:

La codificación consiste en repetir tres veces el mensaje, poniendo un 0 de separación entre cada repetición.

Para decodificar, recuperamos el valor de n e identificamos qué bits eran originalmente los 0 de separación. Entonces:

- Si los dos bits son ahora 1, el intervalo invertido cubre totalmente la segunda repetición del mensaje, por tanto la respuesta es el segundo bloque invertido.
- Si uno de los dos bits es 0, podemos saber que la primera o la última repetición no ha sido alterada, por lo que damos ese bloque como respuesta.
- Si los dos bits son 0, entonces el intervalo invertido está estrictamente dentro de uno de los bloques, por tanto quedan dos bloques iguales entre sí que serán iguales al mensaje.

Esta solución ya proporciona $99{,}06$ puntos y es la mejor solución "sencilla" que hemos encontrado. Era la principal solución que esperábamos de los concursantes durante el concurso.

Solución en menos bits:

Primero, reducimos el problema a otro problema más conocido: el de codificar una cadena binaria de forma que se puedan corregir un pocos errores en posiciones individuales.

Dada una cadena binaria S de longitud n, podemos considerar su correspondiente *cadena de diferencias consecutivas* $\Delta(S)$ de longitud $n - 1$, donde el i-ésimo bit de $\Delta(S)$ es 1 si el i-ésimo y el $(i + 1)$-ésimo bit de S son diferentes, y 0 si son iguales. Nótese que invertir un intervalo de S equivale a invertir dos bits de $\Delta(S)$. Por tanto, podemos reducir el problema a encontrar una codificación C de S que sea descodificable aunque haya dos errores en posiciones aleatorias (la codificación que enviaremos será C' tal que $\Delta(C') = C$). Una codificación sencilla para este caso sería repetir cada bit 5 veces, lo que nos da una solución de longitud $5n + 1$.

La teoría de los códigos correctores de errores nos da varias opciones muy sofisticadas para construir una codificación C con longitud menor a $2n$ que funciona para cualquier error que invierta dos posiciones. Sin embargo, dado que en este problema sólo es necesario corregir errores en posiciones aleatorias con una probabilidad de acierto razonable, se pueden hacer cosas más "chapuceras". En particular, para n grande podemos dividir el mensaje en trozos pequeños, codificar cada trozo por separado con un código corrector de un error, y suponer que no habrán dos errores en el mismo trozo. Hay construcciones sencillas de códigos correctores de un solo error, pero también podemos hacer una construcción pseudoaleatoria, como se muestra en el código.

```cpp
#include<bits/stdc++.h>

using namespace std;

string xor_encode(string s) {
    string t;
    t.push_back(s[0]);
    int n = s.length();
    for (int i=0; i < n; ++i) {
        t.push_back(s[i] == t[i] ? '0' : '1');
    }
    return t;
}

string xor_decode(string s) {
    string t;
    int n = s.length();
    for (int i=0; i+1 < n; ++i) {
        t.push_back('0'+((s[i]-'0')^(s[i+1]-'0')));
    }
    return t;
}

string repetition_encode(string s) {
    const int B = 5;
    string t;
    for (char c : s) {
        for (int i=0; i < B; ++i) t.push_back(c);
    }
    return t;
}

string repetition_decode(string s) {
    const int B = 5;
    string t;
```

```
    vector<int> ct(2);
    for (char c : s) {
        ct[(c-'0')]++;
        if (ct[0]+ct[1] == B) {
            t.push_back(ct[0] > ct[1] ? '0' : '1');
            ct = vector<int>(2);
        }
    }
    return t;
}

const int L = 15;
const int B = 8;
vector<string> blocks;
string null_block;

int hamming_distance(string s, string t) {
    int n = s.length();
    assert(n == (int)t.length());
    int d = 0;
    for (int i=0; i < n; ++i) d += s[i] != t[i];
    return d;
}

void random_init(int seed) {
    mt19937 rng(seed);
    null_block = string(L, '0');
    for (int i=0; i < (1<<B); ++i) {
        string s;
        int min_dist;
        do {
            s = string();
            for(int j=0; j < L; ++j) s.push_back('0'+uniform_int_distribution<>(0, 1)(rng));
            min_dist = hamming_distance(s, null_block);
            for (int j=0; j < i; ++j) {
                min_dist = min(min_dist, hamming_distance(s, blocks[j]));
            }
        } while(min_dist < 3);
        blocks.push_back(s);
    }

}

int toint(string s) {
    int r = 0;
    for (int i=0; i < B; ++i) {
        if (s[i] == '1') r |= (1<<i);
    }
    return r;
}

string tostring(int r) {
    string s;
    for (int i=0; i < B; ++i) {
        s.push_back('0'+(r&1));
        r >>= 1;
    }
    return s;
}

string random_encode(string s) {
    int n = s.length();
    string t;
    for (int i=0; (i+1)*B <= n; ++i) {
        string block = s.substr(i*B, B);
        t += blocks[toint(block)];
    }
    t += null_block;
    if (n%B) {
        t += repetition_encode(s.substr(n-n%B));
    }
    return t;
}

string random_decode(string s) {
    int n = s.length();
    string t;
    for (int i=0; (i+1)*L <= n; ++i) {
        string block = s.substr(i*L, L);
        int md = hamming_distance(block, null_block);
        int mb = -1;
        for (int b=0; b < (1<<B); ++b) {
            int d = hamming_distance(block, blocks[b]);
            if (d < md) {
                md = d;
                mb = b;
            }
        }
        if (mb == -1) {
            if ((i+1)*L < n) {
                t += repetition_decode(s.substr((i+1)*L));
            }
            break;
        }
        else {
            t += tostring(mb);
        }
    }
    return t;
}

int main() {
    random_init(1337);
    string name;
    string str;
    cin >> name >> str;
    string out;
    if (name == "ENC") {
        if (str.length() <= 1e3) {
            out = "00000" + repetition_encode(str);
        }
        else {
            out = "11111" + random_encode(str);
        }
        out = xor_encode(out);
    }
    else if (name == "DEC") {
        str = xor_decode(str);
        cerr << str << endl;
        string pref = str.substr(0, 5);
        string suf = str.substr(5);
        int c0 = 0;
        for (char c : pref) c0 += (c == '0');
        if (c0 >= 3) {
            out = repetition_decode(suf);
        }
        else {
            out = random_decode(suf);
        }
    }
    cout << out << endl;
}
```