TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN CUỐI KÌ

MÔN CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT IT003

ĐỀ TÀI: PHÁ MÃ MẬT KHẨU ĐƯỢC MÃ HÓA BẰNG SUBSET SUM

Sinh viên thực hiện:

- Lê Mậu Anh Phong

MSSV: 21520087

Lớp: ATTN2021

Người hướng dẫn:

- Nguyễn Thanh Sơn
- Nguyễn Đức Vũ

Thành phố Hồ Chí Minh – Năm 2022

I. Giới thiệu:

Mật khẩu là một tài nguyên quan trọng của mỗi cá nhân hay tổ chức. Nó được sử dụng để truy cập vào các tài nguyên quan trọng đối với từng mục tiêu cụ thể. Vì vậy việc bảo vệ mật khẩu là vô cùng cần thiết. Về phía hệ thống, khi nhận được tên đăng nhập và mật khẩu thì hệ thống cần phải lưu vào đâu đó có thể là cơ sở dữ liệu. Tuy nhiên mật khẩu trước khi được lưu cần phải được mã hóa nhằm tránh một người nào đó có quyền truy cập vào cơ sở dữ liệu có thể đọc được. Khi người dùng gửi mật khẩu lên để đăng nhập, hệ thống sẽ mã hóa rồi kiểm tra xem có giống với kết quả lúc đăng kí không. Nếu trùng khớp, người dùng sẽ được phép truy cập vào hệ thống.

Bước mã hóa thường có một số tính chất sau:

- Hàm mã hóa có thể thực hiện trong thời gian đa thức.
- "Không thể" đảo ngược được quá trình mã hóa. "Không thể" ở đây có thể là rất khó, tốn rất nhiều tài nguyên và thời gian để đảo ngược.
- Hạn chế việc xảy ra đụng độ. Việc hai mật khẩu khác nhau đi qua hàm mã hóa cho ra kết quả giống nhau được gọi là đụng độ.

Một bài toán ứng dụng trong việc xây dựng các thuật toán phổ biến được nghiên cứu rất nhiều đó là bài toán subset-sum. Bài toán được phát biểu như sau: Cho n số nguyên $s_0, s_1, \dots, s_{n-1}$ mỗi số có độ dài n bit và số nguyên T. Tìm tập $S \subseteq \{0,1,2,\dots,n-1\}$ sao cho $\sum_{i \in S} s_i = T \ (mod\ 2^n)$.

Sau đây ta sẽ tìm hiểu một số vấn đề trong bài toán đó.

II. Nội dung:

1. Đặt vấn đề:

#	8-char key	40-bit key
0	4szcirzz	1111010010110010001001000100011100111001
1	ibsalkc2	0100000001100100000001011010100001011100
2	usnfujjz	1010010010011010010110100010010100111001
3	bu4mmejr	0000110100111100110001100001000100110001
4	5bd5qclw	1111100001000111111110000000100101110110
5	5vkmuyjo	1111110101010100110010100110000100101110
6	v521cfzw	1010111111111001101100010001011100110110
7	ty12gjq2	1001111000110111110000110010011000011100
8	35b5hs0a	1110111111000011111100111100101101000000
9	0q0w2nna	1101010000110101011011100011010110100000
10	uwnzdxew	1010010110011011100100011101110010010110
11	xmjjhv3s	1011101100010010100100111101011110110010
12	lb3ffann	0101100001111010010100101000000110101101
13	za5lkafk	1100100000111110101101010000000010101010
14	dniohm4x	0001101101010000111000111011001111010111
15	20uqpma2	1110011010101001000001111011000000011100
16	4ac0ist0	1111000000000101101001000100101001111010
17	2c24x3h2	1110000010111001111010111111010011111100
18	c4u2phiq	0001011110101001110001111001110100010000
19	zdvpqfb0	1100100011101010111110000001010000111010

20	wiszzamb	10110010001001011001110010000000110000001
21	0nbn5c0v	11010011010000101101111110000101101010101
22	aipxjpv4	00000010000111110111010010111110101111110
23	iphcrucy	01000011110011100010100011010000001011000
24	3leopdkz	11101010110010001110011110001101010011001
25	3wzkljly	11101101101100101010010110100101011111000
26	robtlkyw	10001011100000110011010110101011000010110
27	h0bjkm2m	00111110100000101001010100110011100001100
28	nniflpbf	01101011010100000101010110111100000100101
29	wvo3no2d	10110101010111011101011010111011100000011
30	kg4ts5y2	01010001101111010011100101111111000011100
31	b12qgbzu	00001110111110010000001100000111001101000
32	wwahbmfq	10110101100000000111000010110000010110000
33	fu15rfbv	00101101001101111111100010010100000110101
34	hv2as24l	00111101011110000000100101110011111001011
35	vw3dilsf	10101101101110100011010000101110010000101
36	1dcqok04	11011000110001010000011100101011010111110
37	owryddqr	01110101101000111000000110001110000010001
38	kvjwuzgn	01010101010100110110101001100100110011101
39	ieae2oph	01000001000000000100111000111001111001111

Bảng 1. Ví dụ về bảng T cho mật khẩu có độ dài là 8 sử dụng bảng chữ cái 32 ký tự.

Cho một bảng T gồm N số nguyên dương với chiều dài n bit. Hệ thống mã hóa luôn được cung cấp mật khẩu với chiều dài cụ thể N bit. Để mã hóa mật khẩu, hệ thống mã hóa sử dụng mật khẩu để chọn các số nguyên trong bảng T và tính tổng của chúng sau đó chia lấy dư cho 2^N sẽ được kết quả mã hóa. Chúng ta sẽ chọn dòng thứ i trong bảng T tương ứng nếu bit thứ i trong mật khẩu được bật.

Trong đề tài này, chúng ta sẽ sử dụng bảng chữ cái gồm 32 ký tự ("abcdefghijklmnopqrstuvwxyz012345") trong các mật khẩu và mã hóa chúng bằng số nguyên 5-bit với 0 mã hóa 'a', 1 mã hóa 'b', và tương tự cho các ký tự tiếp theo. Như vậy, theo quy ước ban đầu, $N = 5C$ trong đó C là số ký tự trong mật khẩu.

Chúng ta sẽ làm quen bằng ví dụ sau. Chúng ta sẽ sử dụng bảng T như đã trình bày trong Bảng 1. Đầu tiên, chúng ta sẽ mã hóa mật khẩu "aaaaaaaa", lấy từng ký tự của nó đổi sang hệ nhị phân rồi nối lại với nhau ta có 00000000000000000000000000000000000000000 sẽ được dạng nhị phân của "aaaaaaaa". Như vậy không có bit nào trong "aaaaaaaa" được bật nên "aaaaaaaa" được mã hóa thành "aaaaaaaa". Chúng ta sẽ thử tiếp với "bbbbbbbb", mật khẩu này được biểu diễn trong hệ nhị phân là 0001000010000100001000010000100001000010. Như vậy các bit thứ 4, 9, 14, 19, 24, 29, 34, 39 (tính từ trái sang) trong "bbbbbbbb" được bật. Cho nên chúng ta sẽ lấy dòng thứ 4, 9, 14, 19, 24, 29, 34, 39 trong bảng T tính tổng và chia lấy dư cho 2^{40} như sau:

#	8-char key	40-bit key
4	5bd5qclw	1111100001000111111110000000100101110110
9	0q0w2nna	1101010000110101011011100011010110100000
14	dniohm4x	0001101101010000111000111011001111010111
19	zdvpqfb0	1100100011101010111110000001010000111010

24	3leopdkz	111010101100100011100111100011010101100 1
29	wvo3no2d	101101010101110111010110101110111000001 1
34	hv2as24l	001111010111100000001001011100111100101 1
39	ieae2oph	010000010000000001001110001110011110011 1
	z3mfp5nv	100110011110101100001010111111110110110101

Như vậy, sử dụng bảng T như đề cập trong Bảng 1, mật khẩu "bbbbbbbb" sẽ được mã hóa thành "z3mfp5nv".

Vậy là tôi đã trình bày xong về cách mã hóa. Các yêu cầu đặt ra:

- Viết chương trình đầu nhận vào mật khẩu đã được mã hóa gồm C ký tự (C có giá trị tối đa là 12) sử dụng bảng chữ cái 32 ký tự theo phương pháp mô tả ở trên. Ngoài ra, chương trình còn được cho biết trước bảng T gồm $N = 5C$ dòng, mỗi dòng là một N-bit key nào đó. Phân tích độ phức tạp của chương trình theo C. Đầu ra của chương trình là mật khẩu trước khi được mã hóa.
- Việc sinh bảng T không tốt có thể dẫn tới tồn tại trường hợp hai mật khẩu khác nhau có cùng có cùng mật khẩu được mã hóa. Trình bày một hướng giải quyết để tạo bảng T sao cho không còn tồn tại trường hợp đó nữa.

2. Giải quyết yêu cầu đặt ra:

a. Yêu cầu thứ nhất:

Nhận xét: Việc mã hóa thực chính là việc chọn chọn ra một tập con của của bảng rồi tính tổng $mod\ 2^N$. Còn việc giải mã thực chất chính là việc tìm tập con có tổng $mod\ 2^N$ bằng một số cho trước. Việc phá mã được coi là hoàn tất khi tìm thấy một tập con thỏa mãn.

Quy ước:

- Đặt $c = C$.
- Ta chỉ sử dụng dạng biểu diễn nhị phân nên bảng T lúc này chỉ cần sử dụng đến cột thứ 2, nên sẽ dùng mảng s để chỉ cột thứ 2 của bảng T.
- Mật khẩu cần giải mã được đặt là $hashed\_message$, dạng nhị phân của $hashed\_message$ kí hiệu là $hash\_num$. Với ví dụ ở trên thì $hashed\_message$ = "z3mfp5nv" và $hash\_num = 890540391861 = 1100111101011000010101111111110110110101_2$.
- $MOD = 2\text{^}N$.

Bài toán phá mã có thể được phát biểu lại như sau: Cho mảng s gồm N số nguyên và số nguyên $hash\_num$, tìm một tập con của s sao cho tổng bằng $hash\_num$ ($mod\ MOD$).

Các bước chính cần làm là:

Bước 1: Đọc vào input.

Bước 2: Tạo mảng s từ bảng T.

Bước 3: Tính $hash\_num$ từ $hashed\_message$.

Bước 4: Tìm tập con F của s mà tổng $mod\ MOD$ bằng $hash\_num$, nếu có thì lưu lại tập con đó và chuyển sang bước 5, ngược lại thử tập con tiếp theo.

Bước 5: Từ kết quả lưu lại chuyển thành mật khẩu.

Các bước 1, 2, 3, 5 tương đối đơn giản, nên sẽ không được trình bày. Phần viết dưới sẽ trình bày cách kiểm tra F có tồn tại không, từ đó tạo cơ sở để có thể truy vết được kết quả.

Bài toán này với phiên bản không có MOD đã được nghiên cứu từ rất lâu và có nhiều thuật toán để giải. Sau khi tham khảo thì tôi thấy có một vài thuật toán:

Thuật toán	Độ phức tạp thời gian	Độ phức tạp không gian
Horowitz and Sahni	$O\left(2^{\frac{N}{2}}.\frac{N}{2}\right)$	$O\left(2^{\frac{N}{2}}\right)$
Schroeppel and Shamir	$O\left(2^{\frac{N}{2}}.\frac{N}{4}\right)$	$O\left(2^{\frac{N}{4}}\right)$

Schroeppel and Shamir là phù hợp nhất điều kiện của đề bài, dưới đây tôi xin trình bày hướng giải dựa trên Schroeppel and Shamir để giải bài toán có MOD.

Ý chính của giải thuật:

Bước 1: Chia s thành 4 phần như sau: đặt $q = \left[\frac{N}{4}\right]$, $h = \left[\frac{N}{2}\right]$

- $L_1 = \{s_0, \dots, s_q\}$
- $L_2 = \{s_{q+1}, \dots, s_h\}$
- $R_1 = \{s_{h+1}, \dots, s_{h+q}\}$
- $R_2 = \{s_{h+q+q}, \dots, s_{N-1}\}$

Bước 2: Sử dụng đệ quy quay lui để tìm tổng và cách chọn của từng tập con ($mod\ MOD$) trong L_1 lưu vào mảng V_0.

Làm tương tự lần lượt với L_2, R_1, R_2 tương ứng với V_1, V_2, V_3.

Khi tạo V_0, V_1, V_2, V_3 thì lưu theo dạng $\{sum, state\}$ với sum là tổng và $state$ là chuỗi bit ở hệ thập phân biểu thị cách chọn của tập hợp để truy vết. Với cách lưu này khi ta cần kết hợp $x \in V_0$, $y \in V_1$ thì sẽ có là $\{x.sum + y.sum, x.state + y.state\}$, tương tự đối với các tập khác.

Bước 3: Sắp xếp V_i tăng dần theo sum.

Bước 4: Giải bài toán: tìm $x \in V_0,\ y \in V_1,\ z \in V_2,\ t \in V_3$ mà $x.sum + y.sum + z.sum + t.sum = hash\_num\ (mod\ MOD)$ ($x.sum$ tức là lấy phần sum của biến x). Giả sử tìm được thì ta có cách chọn thỏa mãn cả đề bài sẽ là $x.state + y.tate + z.state + t.state$.

Để giải quyết được vấn đề ở Bước 3, đầu tiên ta cần xây dựng một cấu trúc dữ liệu priority queue đặc biệt:

Cho hai tập $A, B \subseteq$ Z và $C = A + B = \{a + b \mid a \in A, b \in B\}$

Gọi $c_1, \dots, c_m$ là các phần tử của C theo thứ tự tăng dần. Ta dễ dàng thấy rằng m lên tới $|A|.|B|$. Cấu trúc dữ liệu cần xây dựng gọi là `inc` (đầu vào là hai tập A, B) hỗ trợ truy vấn `inc.next()` mà với lần gọi thứ i ($1 \le i \le m$) sẽ cho ta giá trị của c_i, với $i > m$ thì trả về `EMPTY`. Hơn nữa không gian lưu trữ của nó là $O(|A| + |B|)$. Mã giả cách xây dựng như sau:

Preprocessing `inc` with two sets *A*, *B* as input:

1. Sort $A = a_1, \dots, a_k$, $B = b_1, \dots, b_l$.
2. Initialize priority queue Q.
3. For every $b_j \in B$ add (a_1, b_j) to Q with priority $a_1 + b_j$.

Operation `inc.next()`:

1. **if** Q is empty **then return** `EMPTY`.
2. Let w be the lowest priority in the queue Q.
3. **while** the lowest priority in Q is w **do**
4. | Let (a_i, b_j) be the element with the lowest priority in queue and remove it.
5. | **if** $i < k$ then add (a_{i+1}, b_j) with priority $a_{i+1} + b_j$ to Q.
6. **return** w

Với mã giả trên, output có theo thứ tự sẽ là một dãy tăng dần (Lưu ý là độ ưu tiên khi thêm vào hàng đợi ưu tiên không được mod vì nếu mod output sẽ không còn một dãy tăng dần). Bước khởi tạo tốn thời gian $O(|B|)$. Ta thấy rằng sau bước khởi tạo $|Q| = |B|$, mỗi lần xóa đi một phần tử trong Q thì ta lại có cơ hội thêm vào nhiều nhất một phần tử

nên $|Q| \leq |B|$ do đó độ phức tạo không gian của `inc` là $O(|B|)$. Ngoài ra, vòng lặp bên trong `inc.next()`lặp m lần là Q sẽ rỗng, mỗi lần lặp thực hiện xóa và thêm vào Q (tốn $O(log_2(|B|))$ cho các thao tác trên priority queue) nên tổng thời gian thực hiện của cấu trúc dữ liệu trên là $O(m.log_2(|B|)) = O(|A|.|B|.log_2(|B|))$.

Với cách làm tương tự ta có thể tạo ra một cấu trúc dữ liệu với output là một dãy giảm dần gọi là `dec`.

Ta sẽ tạo `inc` với đầu vào là V_0 và V_1 để liệt kê các phần tử của $V_0 + V_1$ (theo thứ tự tăng dần) và sử dụng `dec` để liệt kê các phần tử của $V_2 + V_3$ (theo thứ tự giảm dần). Ta cần thay đổi một chút để thêm các $state$ vào.

Tiếp theo với mỗi $t \in V_0 + V_1$ và duyệt qua $b \in V_2 + V_3$ bằng kỹ thuật hai con trỏ, đặt $g_1 = t.sum$, $g_2 = b.sum$, tính $g_1 + g_2$ và so sánh với $hash\_num$. Nếu $g_1 + g_2 = hash\_num$ thì có luôn cách giải (cách chọn $t.state + b.state$, không cần tìm nữa), nếu $g_1 + g_2 > hash\_num$ thì tiếp tục gán $g_2 \coloneqq dec.next()$. Nếu `inc` hoặc `dec` rỗng mà vẫn chưa tìm được cách giải thì không có cách giải nào thỏa mãn.

Tuy nhiên vẫn chưa dừng lại ở đó, ta nhớ rằng cái ta thực sự cần là $g_1 + g_2 = hash\_num \pmod{MOD}$ chứ không phải là $g_1 + g_2 = hash\_num$ (không có mod). Vì vậy ta cần biến đổi từ dạng có mod sang dạng không có mod:

$$g_1 + g_2 = hash\_num + u.MOD \text{ (với số nguyên } u \text{ nào đó).}$$

Vì g_1, g_2 là tổng hai số nguyên không âm nhỏ hơn MOD, nên tối đa $g_1 = g_2 = MOD - 1 + MOD - 1 = 2MOD - 2$. Nên tối đa $g_1 + g_2 = 4MOD - 4 < 4MOD$. Nên ta có u là số nguyên không âm nhỏ hơn 4. Do đó sau khi thực hiện thuật toán trên với $hash\_num$, ta gán $hash\_num \coloneqq hash\_num + MOD$ và thực hiện ba lần như vậy.

Dưới đây là cài đặt chi tiết để tìm một nghiệm của cả yêu cầu thứ nhất, nếu muốn ghi ra tất cả các nghiệm thì cần sửa lại một vài chỗ (nhường lại cho bạn đọc).

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

using namespace std;

const string ALPHABET = "abcdefghijklmnopqrstuvwxyz012345";

int n;
long long hash_num = 0, s[100], MOD;
string hashed_message;
vector <pair <long long, long long> > V[4];

bool getBit(long long v, int p)
{
    return (v >> p) & 1;
}

long long turnOn(long long v, int p)
{
    return (1LL << p) | v;
}

void input()
{
    cin >> n;
    n *= 5;
    MOD = 1LL << n;
    long long sum = 0;
    for (int i = 0; i < n; ++i)
    {
        string x;
```

```
        cin >> x;
        string y = "";
        for (auto c: x)
        {
            int v = ALPHABET.find(c);
            for (int j = 4; j >= 0; --j)
                y += (char)(48 + getBit(v, j));
        }

        reverse(y.begin(), y.end());
        for (int j = 0; j < n; ++j)
            if (y[j] == '1')
                s[i] += 1LL << j;
    }
}

class inc
{
    private:
        vector <pair <long long, long long> > *a, *b;
        priority_queue <pair <pair <long long, long long>, pair <int, int> > > pq;

    public:
        inc(vector <pair <long long, long long> > *a, vector <pair <long long, long long> >
*b)
        {
            this->a = a;
            this->b = b;
            for (int i = 0; i < b->size(); ++i)
                pq.push({{- ((*a)[0].first + (*b)[i].first), (*a)[0].second +
(*b)[i].second}, {0, i}});
        }

        pair <long long, long long> top()
        {
            if (pq.empty()) return {LLONG_MAX, LLONG_MAX};
            long long w = pq.top().first.first;
            long long state = pq.top().first.second;
            while (pq.size() && pq.top().first.first == w)
            {
                auto p = pq.top(); pq.pop();
                int i = p.second.first, j = p.second.second;
                if (i < a->size() - 1)
                    pq.push({{- ((*a)[i + 1].first + (*b)[j].first), (*a)[i + 1].second +
(*b)[j].second}, {i + 1, j}});
            }

            return {- w, state};
        }
};
```

```
void calc_hash_num()
{
    cin >> hashed_message;
    reverse(hashed_message.begin(), hashed_message.end());
    long long pw32 = 1;
    for (int i = 0; i < n/5; ++i)
    {
        int k = ALPHABET.find(hashed_message[i]), x = 0;
        hash_num += k * pw32;
        pw32 *= 32LL;
    }
}

void backtrack(int k, int r, long long sum, long long state, int id)
{
    if (k > r)
        return V[id].push_back({(id < 2 ? 1 : - 1) * sum, state}), void();

    for (int i = 0; i < 2; ++i)
        if (i)
            backtrack(k + 1, r, (sum + s[k]) % MOD, turnOn(state, k), id);
        else
            backtrack(k + 1, r, sum, state, id);
}

void convert(long long stateX)
{
    string ans = "";
    for (int i = 0; i < n/5; ++i)
    {
        int order = 0;
        for (int j = 5 * i, k = 4; j < 5 * (i + 1); ++j, --k)
            order += getBit(stateX, j) * (1 << k);
        ans += ALPHABET[order];
    }

    cout << ans;
    exit(0);
}

void two_pointers(long long hash_num)
{
    inc lPart(&V[0], &V[1]), rPart(&V[2], &V[3]);
    pair <long long, long long> last = {- 1, - 1};
    while (1)
    {
        auto l = lPart.top(), r = last;
        if (r.first == - 1) last = r = rPart.top();
        if (l.first == LLONG_MAX || r.first == LLONG_MAX)
```

```
            break;

        while (1)
        {
            if (r.first == LLONG_MAX) return;
            if (hash_num - l.first == - r.first) convert(l.second + r.second);
            else if (hash_num - l.first > - r.first)
            {
                last = r;
                break;
            }
            last = r = rPart.top();
        }
    }
}

int main()
{
    freopen("input.txt", "r", stdin);

    input();
    calc_hash_num();

    backtrack(0, n / 4 - 1, 0, 0, 0);
    backtrack(n / 4, n / 2 - 1, 0, 0, 1);
    backtrack(n / 2, n / 2 + n / 4 - 1, 0, 0, 2);
    backtrack(n / 2 + n / 4, n - 1, 0, 0, 3);

    for (int i = 0; i < 4; ++i) sort(V[i].begin(), V[i].end());

    two_pointers(hash_num);
    two_pointers(hash_num + MOD);
    two_pointers(hash_num + 2 * MOD);
    two_pointers(hash_num + 3 * MOD);

    cout << - 1;
    return 0;
}
```

Phân tích độ phức tạp:

Phần `input()`: duyệt N lần để tạo mảng s, mỗi lần lại duyệt đọc vào một xâu C ký tự, chuyển xâu đó sang nhị phân (N bit), rồi từ dạng nhị phân đó chuyển qua thập phân (bước này tốn $O(N)$) lưu vào mảng s nên độ phức tạp thời gian là $O(N^2)$ và không gian là $O(N)$.

Phần `calc_hash_num()`: duyệt qua từng ký tự của mật khẩu cần phá mã nên độ phức tạp thời gian là $O(c)$, độ phức tạp không gian là $O(1)$ vì chỉ lưu lại một vài biến thông thường.

Phần `backtrack()`: duyệt qua từng tập con của tập cha mà tập cha có số lượng phần tử không quá $\frac{N}{4}$ nên tốn thời gian $O\left(2^{\frac{N}{4}}\right)$. Mỗi tập con cần lưu lại thông tin thành mảng V_i cho các bước sau nên tốn về không gian $O\left(\frac{N}{4}\right)$.

Phần `sort()`: dùng để sắp xếp lại mảng sau bước `backtrack()` nên về thời gian cần $O\left(2^{\frac{N}{4}}.log_2\left(2^{\frac{N}{4}}\right)\right) = O\left(2^{\frac{N}{4}}.\frac{N}{4}\right)$, chúng ta không dùng cấu trúc dữ liệu để lưu lại gì ở bước này nên về không gian là $O(1)$.

Phần `two_pointers()`: tạo ra `inc` và `dec` cần thời gian $O\left(2^{\frac{N}{4}}.\frac{N}{4}\right)$ và không gian $O\left(2^{\frac{N}{4}}\right)$ (bước Preprocessing). Duyệt hai con trỏ (trình bày ở phần trước) qua từng phần tử của `inc` và `dec` mỗi phần tử một lần. Mà `inc` và `dec` được xây dựng là hai mảng có số lượng phần tử không quá $2^{\frac{N}{4}}$ nên `inc` và `dec` có số lượng phần tử tối đa khoảng $2^{\frac{N}{2}}$ nên cần $O\left(2^{\frac{N}{4}}.\frac{N}{4}\right)$. Tại bước này không lưu lại gì – độ phức tạp không gian $O(1)$.

Vì vậy ta có độ phức tạp của cả thuật toán về mặt thời gian là $O\left(2^{\frac{N}{4}}.\frac{N}{4}\right) = O\left(2^{\frac{5C}{4}}.\frac{5C}{4}\right)$, không gian là $O\left(2^{\frac{N}{4}}\right) = O\left(2^{\frac{5C}{4}}\right)$.

Thử nghiệm thực tế: thực hiện trên máy 4 GB RAM. (Test)

Thứ tự	Độ dài mật khẩu	Thời gian phá mã	Không gian cần dùng
Test 1	$C = 5$	< 0.01 s	< 1 KiB
Test 2	$C = 6$	< 0.01 s	< 1 KiB
Test 3	$C = 8$	0.69 s	3060 KiB
Test 4	$C = 10$	31.18 s	6436 KiB
Test 5	$C = 12$	1246.09 s	6436 KiB
Test 6	$C = 8$	0.75 s	2128 KiB

b. Yêu cầu thứ hai:

Dạng nhị phân của mật khẩu sau mã hóa chính là kết quả của việc tính tổng (mod MOD) của một tập con của mảng s. Vì vậy yêu cầu của câu này tương đương với việc giải bài toán: Xây dựng một mảng s gồm N phần tử sao cho tổng mọi tập con của s (mod MOD) đôi một khác nhau. Nên tôi sẽ trình bày cách giải bài toán này thay cho bài toán gốc.

Ý tưởng ban đầu là xây dựng mảng s với $s_i = 2^i (0 \le i < N)$. Lúc này với cách chọn được biểu diễn bằng dãy bit $x_{N-1}, x_{N-2}, \dots, x_1, x_0$ sẽ có tổng được biểu diễn dưới dạng nhị phân cũng chính là $x_{N-1}, x_{N-2}, \dots, x_1, x_0$.

Ví dụ: $N = 4$, $s = [1, 2, 4, 8]$. Nếu ta chọn tập được biểu diễn là 0111 thì sẽ có tổng là 7 (0111)

Nên sẽ có 2^N tập có tổng tương ứng với chính cách chọn. Đã thỏa mãn được đề bài nhưng trên thực tế việc sử dụng mảng s có tính chất $s_i \ge 2s_{i-1}$ rất dễ bị bẻ khóa kỹ thuật toán tham lam (chi tiết sẽ không được trình bày ở đây). Nên cần tìm cách cải tiến, cách của tôi như sau:

Bằng thực nghiệm trên một vài ví dụ tôi thấy rằng mình có thể cải tiến bằng cách xây dựng một mảng k với k_i là một số nguyên không âm, rồi gán $s_i = 2^i + k_i * 2^{i+1}$ mod MOD. Tránh chọn $k_i = 0$ với mọi i vì sẽ rơi vào cách tiếp cận ban đầu.

Lấy ví dụ với $N = 4$, với cách tiếp cận $s_i = 2^i$, ta xếp các tập con lên bảng:

Cách chọn	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Tổng	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Với cách tiếp cận thứ hai, bản chất chính là biến 2^i thành $2^i + k_i 2^{i+1} (mod\ MOD)$ nên nếu ở cách tiếp cận ban đầu có tổng là p mà p lại có có 2^i trong đó (ví dụ 0111 thì có 2^0, 2^1, 2^2 ở trong) thì sau phép biến đổi p trở thành $p + k_i 2^{i+1}$ với mọi i.

Ví dụ có:

i	3	2	1	0
k_i	0	0	0	1
$k_i 2^{i+1}$	0	0	0	2

s_i	8	4	2	3

Vì k_1, k_2, k_3 đều bằng 0 nên không thay đổi gì, nên sẽ chỉ xét k_0, những số được tô tím là số có 2^0 trong biểu diễn nhị phân:

Cách chọn	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Trước	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
	Những số màu tím là những số có 2^0 trong cách biểu diễn nhị phân. Toàn bộ số màu tím sau biến đổi được cộng thêm $k_0 * 2^{(0+1)} = 2$. Còn số màu đen giữ nguyên.															
Sau	0000	0011	0010	0101	0100	0111	0110	1001	1000	1011	1010	1101	1100	1111	1110	0001

Ta sẽ thử với ví dụ phức tạp hơn:

i	3	2	1	0
k_i	0	1	0	2
$k_i 2^{i+1}$	0	8	0	4
s_i	8	10	2	5

Với bảng này thì ta có những điều sau:

- Những số có 2^0 trong biểu diễn nhị phân của nó sẽ được cộng thêm $k_0 2^{(0+1)} = 4$ (mod MOD).
- Những số có 2^2 trong biểu diễn nhị phân của nó sẽ được cộng thêm $k_2 2^{(2+1)} = 8$ (mod MOD).
- Những số còn lại đứng yên.

Vậy với những số cả cả 2^0 và 2^2 trong biểu diễn nhị phân sẽ chịu ảnh hưởng từ cả hai điều đầu tiên (cộng $4 + 8$). Tóm lại:

- Những số có 2^0 mà không có 2^2 trong biểu diễn nhị phân của nó (màu tím) sẽ được cộng thêm 4 lần (mod MOD).
- Những số có 2^2 mà không có 2^0 trong biểu diễn nhị phân của nó (màu xanh) sẽ được cộng thêm 8 (mod MOD).
- Những số có cả 2^0 và 2^2 trong biểu diễn nhị phân của nó (màu vàng) sẽ được công thêm $4 + 8 = 12$ (mod MOD).
- Những số còn lại giữ nguyên.

Cách chọn	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Trước	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Sau	0000	1101	0010	1111	1100	1001	1110	1011	1000	0101	1010	0111	0100	0001	0110	0011

Sau phép biến đổi đó không có 2 số nào nằm vào cùng một ô, tức là các tổng của tập con của s đôi một khác nhau.

Tôi có suy đoán rằng: Việc cộng thêm đó làm cho các dải màu di chuyển sang trái theo vòng tròn theo một quy tắc nào đó. Sử dụng ví dụ 2 ta thấy:

- Dải màu tím được dịch trái 4 lần.
- Dải màu xanh được dịch trái 8 lần.
- Dải màu vàng được dịch sang trái 12 lần.
- Dải màu đen được dịch trái 0 lần.
- Các dải sau khi dịch không "đè" lên nhau.

Tuy nhiên lời giải thích đó vẫn chưa thuyết phục, sau đây là phần trình bày chứng minh bằng toán học:

Gọi P_1 và P_2 là hai vị trí bất kì trước khi biến đổi.

S_1 là tập hợp những vị trí chứa bit 1 trong dạng biểu diễn nhị phân của P_1.

S_2 là tập hợp những vị trí chứa bit 1 trong dạng biểu diễn nhị phân của P_2.

P_{x1} là P_1 sau biến đổi.

P_{x2} là P_2 sau biến đổi.

Ta có: $P_1 = \sum_{x \in S_1} 2^x$

$P_2 = \sum_{x \in S_2} 2^x$

$P_{x1} = \sum_{x \in S_1} 2^x + k_x 2^{x+1} = \sum_{x \in S_1} 2^x(1 + 2k_x)$

$P_{x2} = \sum_{y \in S_2} 2^y + k_y 2^{y+1} = \sum_{y \in S_2} 2^y(1 + 2k_y)$

Ta cần chứng minh: nếu $P_{x1} = P_{x2}$ thì $P_1 = P_2$ (hoặc $S_1 = S_2$)

$P_{x1} = P_{x2} (mod\ MOD)$

$\Leftrightarrow \sum_{x \in S_1} 2^x(1 + 2k_x) = \sum_{y \in S_2} 2^y(1 + 2k_y)\ (mod\ MOD)$

$\Leftrightarrow \sum_{x \in S_1} 2^x(1 + 2k_x) = \sum_{y \in S_2} 2^y(1 + 2k_y) + u.MOD$ (luôn tồn tại u thỏa mãn)

$\Leftrightarrow \sum_{x \in S_1} 2^x(1 + 2k_x) = \sum_{y \in S_2} 2^y(1 + 2k_y) + u.2^N$ (1)

Trường hợp 0: Nếu $S_1 = S_2$, không cần chứng minh thêm.

Trường hợp 1: Nếu ($S_1 = \emptyset$ hoặc $S_2 = \emptyset$) và $S_1 \neq S_2$

Không mất tính tổng quát, giả sử $S_1 = \emptyset$. Ta có $P_1 = P_{1x} = 0$, nên (1) trở thành

$0 = \sum_{y \in S_2} 2^y(1 + 2k_y) + u.2^N$

$\Leftrightarrow 0 = 2^m(1 + 2k_m) + \sum_{y \in S_2 \setminus \{m\}} 2^y(1 + 2k_y) + u.2^N$ (với m là phần tử nhỏ nhất trong S_2)

Chia cả hai vế cho 2^m, ta được:

$0 = 1 + 2k_m + \sum_{y \in S_2 \setminus \{m\}} 2^{y-m}(1 + 2k_y) + u.2^{N-m}$

Do $y - m \geq 1$ và $N - m \geq 1$ nên vế trái sẽ là một số chẵn, còn vế phải sẽ là một số lẻ. Điều này trái với điều kiện của đề bài nên không thể xảy ra.

Trường hợp 2: Các trường hợp còn lại

Gọi m_1 là phần tử nhỏ nhất của S_1.

m_2 là phần tử nhỏ nhất của S_2.

Nếu $m_1 = m_2$, thì ta trừ hai vế của (1) cho $2^{m_1}(1 + 2k_{m_1})$ rồi thay thế S_1 thành $S_1 \setminus \{m_1\}$, S_2 thành $S_2 \setminus \{m_2\}$ và tiếp tục xét từ đầu.

Nếu $m_1 \neq m_2$, không mất tính tổng quát giả sử $m_1 < m_2$, ta có:

(1) $\Leftrightarrow 2^{m_1}(1 + 2k_{m_1}) + \sum_{x \in S_1 \setminus \{m_1\}} 2^x(1 + 2k_x) = \sum_{y \in S_2} 2^y(1 + 2k_y) + u.2^N$

Sau đó, chia hai vế của (1) cho 2^{m_1}, ta có:

$1 + 2k_{m_1} + \sum_{x \in S_1 \setminus \{m_1\}} 2^{x-m_1}(1 + 2k_x) = \sum_{y \in S_2} 2^{y-m_1}(1 + 2k_y) + u.2^{N-m_1}$

Vì $x - m_1 > 0$, $y - m_1 > 0$, $N - m_1 > 0$ nên vế trái là một số lẻ, vế phải là một số chẵn. Do đó không thể xảy ra trường hợp này.

Vì vậy $m_1 = m_2$.

Do đó $S_1 = S_2$ hay $P_1 = P_2$.

Tức là với $P_1 \neq P_2$ thì $P_{1x} \neq P_{2x}$.

Vậy sau phép biến đổi ta có dãy số thỏa mãn đề bài.

Tóm lại ta có một hướng xây dựng mảng s mà tổng mọi tập con của nó đôi một phân biệt là:

$$s_i = 2^i + k_i . 2_{i+1}$$ với k_i là một số nguyên bất kì.

III. Một số vấn đề khác:

- Ta thấy rằng thời gian để phá mã tăng lên rất nhanh khi mật khẩu dài hơn đây chính là cơ sở của khuyến cáo nên đặt mật khẩu phức tạp và dài.

IV. Tài liệu tham khảo:

- Subset sum problem.
- Efficient Cryptographic Schemes Provably as Secure as Subset Sum.
- Improving Schroeppel and Shamir's Algorithm for Subset Sum via Orthogonal Vectors.
- Improving Schroeppel and Shamir's Algorithm for Subset Sum via Orthogonal Vectors (video).

V. Link github: link.