HƯỚNG DẪN GIẢI — Ban Chuyên môn Tin học

Bài 1: Số ô đen

Lời giải

Nhận thấy rằng nếu như $m \cdot n$ là số chẵn, thì số ô đen sẽ = số ô trắng = $\frac{m \cdot n}{2}$. Nếu $m \cdot n$ là số lẻ, thì xét tiếp qua i, j. Nếu i và j đều nằm ở vị trí có thứ tự cùng chẵn hoặc cùng lẻ, thì số ô đen là $\frac{m \cdot n+1}{2}$. Ngược lại số ô đen sẽ là $\frac{m \cdot n-1}{2}$.

Cài đặt (tham khảo)

```cpp
#include <bits/stdc++.h>
using namespace std;
int main()
{
    long long n,m,i,j;
    cin >> n >> m >> i >> j;
    long long ans;
    if ((m * n) % 2 == 0)
        ans = (m * n) / 2;
    else
    {
        if (i % 2 == 0 && j % 2 == 0)
            ans = ((m * n) + 1) / 2;
        else if (i % 2 != 0 && j % 2 != 0)
            ans = ((m * n) + 1) / 2;
        else ans = ((m * n) - 1) / 2;
    }
    cout << ans;
    return 0;
}
```

Bài 2: Hình hộp chữ nhật

Subtask 1: Vét cạn

Vì $n \leq 100$ và ta có thể dễ dàng nhận thấy $x, y, z \leq n$ nên ta sẽ lần lượt thử từng bộ ba số x, y, z và kiểm tra xem nếu $x \cdot y \cdot z = N$ thì chọn bộ 3 có giá trị z lớn nhất. Độ phức tạp $O(n^3)$.

Subtask 2: Vét cạn (optimize)

Tương tự thuật ở subtask 1 nhưng ta có thể thấy rằng chỉ cần có hai giá trị x, y thì ta có thể xác định giá trị $z = \frac{N}{x \cdot y}$. Và vì $N \leq 5000$ nên ta sẽ dùng 2 vòng lặp để duyệt mọi cặp giá trị x, y sao cho N chia hết cho $x \cdot y$ và cực đại hóa giá trị $z = \frac{N}{x \cdot y}$. Độ phức tạp $O(n^2)$.

Subtask 3: Thuật chuẩn

Để một bộ 3 $\{x, y, z \mid x \cdot y \cdot z = N\}$ và z là lớn nhất khi và chỉ khi giá trị $x \cdot y$ là bé nhất.

Từ nhận xét trên ra có thể thấy x sẽ là ước số bé nhất của N, y là ước số bé nhất của giá trị $\frac{N}{x}$, khi đó N sẽ chia hết cho $x \cdot y$ và $x \cdot y$ là sẽ có giá trị bé nhất khi đó $z = \frac{N}{x \cdot y}$ sẽ mang giá trị lớn nhất.

Đưa bài toán về tìm ước bé nhất của một số. Độ phức tạp $O(\sqrt{N})$.

Cài đặt (tham khảo)

```cpp
#include <bits/stdc++.h>
using namespace std;

int32_t main(){
    ios_base::sync_with_stdio(false);
    cin.tie(0);

    int n; cin >> n;

    int a = 1, b = 1;
    for(int i=2; i*i<=n; i++) if(n%i == 0){
        a = i;
        n /= i;
        break;
    }

    for(int i=2; i*i<=n; i++) if(n%i == 0){
        b = i;
        n /= i;
        break;
    }

    if(a == 1 || b == 1 || n == 1) cout << -1;
    else cout << a << ' ' << b << ' ' << n << endl;

    return 0;
}
```

Bài 3: Cắt dãy

Subtask 1: Vét cạn

Ta thấy rằng nếu đặt mỗi phần tử 1 trong 2 trạng thái là: *ghép vào nhóm của phần tử sau* và *kết thúc một nhóm*, và mặc định phần tử cuối sẽ có trạng thái *kết thúc một nhóm*, thì mỗi cách chọn các trạng thái cho $n-1$ phần tử đầu tiên sẽ tương ứng với một cách cắt dãy.

Do giới hạn $n \leq 21$ nên ta hoàn toàn có thể duyệt hết toàn bộ cấu hình có k trạng thái *kết thúc một nhóm* và tính chi phí theo định nghĩa đề bài.

- Độ phức tạp thời gian: $O(2^n)$.

Subtask 2: Quy hoạch động

Gọi $\texttt{dp}[i][j]$ là chi phí tối thiểu để cắt i phần tử đầu tiên của dãy a thành j đoạn. Ta thử hết tất cả các vị trí k là vị trí ngay trước vị trí bắt đầu của đoạn thứ j, nói cách khác, đoạn thứ j sẽ là đoạn từ phần tử thứ $k+1$ đến phần tử thứ i. Khi đó, k phần tử đầu tiên sẽ được cắt thành $j-1$ nhóm. Như vậy, công thức truy hồi là:

$$\texttt{dp}[i][j] = \min_{0 \leq k < i} (\max(\texttt{dp}[k][j-1], (\texttt{pre}[i] - \texttt{pre}[k])^2))$$

với $\texttt{pre}[i]$ là tổng tiền tố độ dài i của dãy a.

- Độ phức tạp thời gian: $O(n^2 k)$.

Subtask 3: Quy hoạch động + CTDL + chặt nhị phân

Dễ thấy, nếu tạo được một cách cắt có chi phí $\leq x$ thì ta cũng tạo được một cách cắt có chi phí $\leq x+1$. Do đó, ta có thể chặt nhị phân trên hàm $\texttt{check}(x)$ kiểm tra xem có tồn tại một cách cắt có chi phí $\leq x$ hay không.

Để xây dựng hàm `check`(x), ta sử dụng lại ý tưởng Quy hoạch động ở subtask 2, như bây giờ ta định nghĩa `dp`$[i][j]$ là một giá trị boolean cho biết có thể cắt i phần tử đầu tiên của dãy a thành j đoạn sao cho chi phí $\leq x$.

Ta thấy `dp`$[i][j] = 1$ khi tồn tại một vị trí k sao cho $(\texttt{pre}[i] - \texttt{pre}[k])^2 \leq x$ và `dp`$[k][j-1]$ cũng bằng 1. Mà để bình phương của một số s bé hơn một cận trên, tức là $s^2 \leq x$, thì ta có $|s| \leq \sqrt{x}$ hay $-\sqrt{x} \leq s \leq \sqrt{x}$. Như vậy, ta cần chọn k sao cho $-\sqrt{x} \leq \texttt{pre}[i] - \texttt{pre}[k] \leq \sqrt{x}$, tức là:

$$\texttt{pre}[i] - \lfloor\sqrt{x}\rfloor \leq \texttt{pre}[k] \leq \texttt{pre}[i] + \lfloor\sqrt{x}\rfloor$$

Để tối ưu Quy hoạch động, ta nén số các giá trị `pre`$[i]$ rồi sử dụng Fenwick Tree để tìm giá trị k cần tìm.

- Độ phức tạp bộ nhớ: $O(nk)$ hoặc $O(n)$ tùy cách cài đặt.
- Độ phức tạp thời gian: $O(\log(a^2n^2) \cdot nk \log n)$.

Cài đặt (tham khảo)

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
typedef long double ld;
typedef pair<ll,ll> pl;
typedef pair<int,int> pii;
typedef tuple<int,int,int> tt;

#define all(a) a.begin(), a.end()
#define filter(a) a.erase(unique(all(a)), a.end())

const int mn = 1e4 + 4;
int a[mn], pre[mn], leftBound[mn], rightBound[mn], pos[mn], n, blocks;
vector<int> cmp;
bool dp[2][mn];

ll floorSqrt (ll n) {
    ll x = sqrt(n), ans = 0;
    for (ll i = x - 2; i <= x + 2; i++)
        if (i * i <= n) ans = i;
    return ans;
}

ll ceilSqrt (ll n) {
    ll x = sqrt(n), ans = 0;
    for (ll i = x + 2; i >= x - 2; i--)
        if (i * i >= n) ans = i;
    return ans;
}

struct BIT {
    vector<int> tr;
    BIT (int sz) : tr(sz + 1) {}

    int p (int k) { return k & -k; }

    void update (int k) {
        for (; k < tr.size(); k += p(k)) tr[k]++;
    }

    int preSum (int k , int ans = 0) {
        for (; k; k -= p(k)) ans += tr[k];
        return ans;
    }

    int query (int l, int r) {
```

```
        return preSum(r) - preSum(l - 1);
    }

    void refresh() { fill(all(tr), 0); }
};

int lowerBound (int targ) {
    return lower_bound(all(cmp), targ) - cmp.begin() + 1; // return 1-indexed value
}

bool check (ll ub) {
    int x = floorSqrt(ub);
    for (int i = 1; i <= n; i++) {
        leftBound[i] = lowerBound(pre[i] - x);
        rightBound[i] = lowerBound(pre[i] + x + 1) - 1;
    }

    for (int t = 0; t < 2; t++)
        for (int i = 0; i <= n; i++) dp[t][i] = 0;
    dp[0][0] = 1;

    BIT tree(n + 1);
    for (int k = 1; k <= blocks; k++) {
        int t = k & 1;
        for (int i = 0; i <= n; i++) dp[t][i] = 0;

        tree.refresh();
        if (dp[t ^ 1][0])
            tree.update(pos[0]);

        for (int i = 1; i <= n; i++) {
            dp[t][i] = tree.query(leftBound[i], rightBound[i]);
            if (dp[t ^ 1][i]) tree.update(pos[i]);
        }
    }
    return dp[blocks & 1][n];
}

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);

    cin >> n >> blocks;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        pre[i] = pre[i - 1] + a[i];
        cmp.push_back(pre[i]);
    }
    cmp.push_back(0);
    sort(all(cmp)), filter(cmp);

    for (int i = 0; i <= n; i++) pos[i] = lowerBound(pre[i]);
    if (check(0)) return cout << 0, 0;

    ll ans = 0;
    for (ll msk = 1LL << 57; msk > 0; msk >>= 1) {
        if (!check(ans | msk)) ans |= msk;
    }
    cout << ans + 1;

    return 0;
}
```

Bài 4: Ước chính phương

Subtask 1: Vét cạn

Tính trước $f(n)$ với mọi $n \in [1; 100]$. Để tính một giá trị $f(n)$ bất kì, ta duyệt mọi cặp ước rồi tính theo định nghĩa trong $O(n^2)$. Với mọi truy vấn, ta chỉ việc duyệt trâu các giá trị từ L đến R rồi đưa ra kết quả.

- Độ phức tạp thời gian: $O(N^3 + T \cdot N)$.

Subtask 2: Sàng nguyên tố

Để một số là số chính phương thì trong phân tích thừa số nguyên tố của nó, mọi số mũ đều là **số chẵn**. Từ đó, ta có thể rút ra nhận xét rằng trong phân tích thừa số của a và b, số mũ của cùng một thừa số nguyên tố phải có cùng tính chẵn/lẻ.

Do các thừa số nguyên tố là độc lập nhau, ta có thể tính cho từng thừa số rồi nhân chúng lại để có đáp án cuối cùng. Cụ thể, với phân tích thừa số nguyên tố $n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_k^{a_k}$, ta có công thức tính $f(n)$:

$$f(n) = \prod_{1 \le i \le k} \texttt{calc}(a_k)$$

với hàm $\texttt{calc}(x)$ là hàm tính số cặp (a, b) có cùng tính chẵn/lẻ sao cho $a, b \le x$. Việc xây dựng công thức tính $\texttt{calc}(x)$ sẽ được nhường lại cho độc giả.

Với công thức này, ta hoàn toàn có thể tính $f(n)$ với mọi $n \in [1; 10^6]$ trong khoảng $O(N \ln N)$, kết hợp thuật toán Sàng nguyên tố để có thể phân tích thừa số nguyên tố một cách hiệu quả.

Để trả lời các truy vấn, ta chỉ việc xây dựng prefix sum cho hàm f rồi tính tổng đoạn $[L; R]$ trong $O(1)$.

- Độ phức tạp thời gian: $O(N \ln N + T)$.

Subtask 3: Sàng nguyên tố mở rộng

Ta có thể tính trực tiếp hàm f bằng một lần chạy Sàng nguyên tố duy nhất. Ý tưởng là với mọi số nguyên tố p, ta cần tính số mũ của nó trong các bội cần duyệt, điều này có thể được thực hiện bằng Quy hoạch động cơ bản.

Ngoài ra, độc giả cũng có thể tham khảo dạng mở rộng của thuật toán Sàng nguyên tố ở Chuyên đề Số nguyên tố sắp tới của dự án Chicken Minds (The Gifted Battlefield).

- Độ phức tạp bộ nhớ: $O(N)$.
- Độ phức tạp thời gian: $O(N \ln \ln N + T)$.

Cài đặt (tham khảo)

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
typedef long double ld;
typedef pair<ll,ll> pl;
typedef pair<int,int> pii;
typedef tuple<int,int,int> tt;

#define all(a) a.begin(), a.end()
#define filter(a) a.erase(unique(all(a)), a.end())

const int mn = 1e7 + 7, MOD = 1e9 + 7;
int sieve[mn], vp[mn];

int add (int a, int b) {
    assert(0 <= a && a < MOD && 0 <= b && b < MOD);
```

```
    return a + b - (a + b < MOD ? 0 : MOD);
}

int sub (int a, int b) {
    assert(0 <= a && a < MOD && 0 <= b && b < MOD);
    return a - b + (a - b >= 0 ? 0 : MOD);
}

int mul (int a, int b) {
    assert(0 <= a && a < MOD && 0 <= b && b < MOD);
    return 1LL * a * b % MOD;
}

int sq (int a) { return mul(a, a); }
int odd (int k) { return sq((k + 1) / 2); }
int even (int k) { return sq((k + 2) / 2); }

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);

    for (int i = 1; i < mn; i++) sieve[i] = 1;
    for (int p = 2; p < mn; p++) {
        if (sieve[p] != 1) continue; // p is composite
        for (int i = 1; p * i < mn; i++) {
            vp[i] = (i % p ? 0 : vp[i / p] + 1);
            sieve[p * i] = mul(sieve[p * i], add(odd(vp[i] + 1), even(vp[i] + 1)));
        }
    }
    for (int i = 1; i < mn; i++)
        sieve[i] = add(sieve[i - 1], sieve[i]);

    int TC; cin >> TC;
    while (TC--) {
        int L, R; cin >> L >> R;
        cout << sub(sieve[R], sieve[L - 1]) << "\n";
    }

    return 0;
}
```

Bài 5: Xâu nhị phân

Subtask 1: Xét trường hợp

Với $n = 3$, ta chỉ cần xét 2 trường hợp như sau:

- Nếu số lượng ký tự `1` trong xâu s là 0 hoặc 3, ta không thể tạo ra hai vị trí khác nhau, như vậy đáp án là 0.
- Nếu số lượng ký tự `1` trong xâu s là 1 hoặc 2, ta cố gắng đưa xâu về dạng `010` hoặc `101`, như vậy đáp án là 1 hoặc 2.

Ý tưởng chung

Trước hết, ta cần xây dựng một công thức để tính số lần hoán đổi tối thiểu để biến đổi một sau nhị phân S thành T.

Rõ ràng, việc hoán đổi hai ký tự giống nhau sẽ làm xâu không thay đổi nhưng lại tốn thêm một thao tác. Do đó, ta chỉ thực hiện các thao tác hoán đổi ký tự `0` với `1` hoặc ngược lại. Từ đó, ta cũng đưa ra nhận xét: ký tự `1` thứ i trong xâu S sẽ tương ứng với ký tự `1` thứ i trong xâu T. Điều này đúng bởi vì nếu ký tự `1` thứ i trong xâu S tương ứng với ký tự `1` thứ j (với $i \neq j$) trong xâu T thì ta đã thực hiện hoán đổi ký tự `1` thứ i với các ký tự `1` nằm giữa i và j, và như đã phân tích, điều này là phung phí. Điều tương tự cũng đúng với ký tự `0`.

Như vậy, để tính số lần hoán đổi, ta chỉ cần quan tâm đến số lần mà các ký tự `1` bị hoán đổi vị trí với ký tự `1`. Kết hợp với nhận xét nêu trên, ta thấy rằng hai ký tự bất kì sẽ bị hoán đổi với nhau khi và chỉ khi:

- Gọi i, j lần lượt là vị trí của chúng trong xâu S, ta có $i < j$.
- Gọi i', j' lần lượt là vị trí của chúng trong xâu T, ta có $i' > j'$.

Như vậy, với mỗi ký tự `1`, ta cần tính chênh lệch của số ký tự `0` nằm ở bên trái nó ở xâu S và T. Tức là ta có công thức tính số lần hoán đổi như sau:

$$\sum_{1 \le i \le n, S[i]=1} |\texttt{countZero}_S[i] - \texttt{countZero}_T[\texttt{pos}[i]]| = \sum_{1 \le i \le n, S[i]=1} |i - \texttt{pos}[i]|$$

với $\texttt{countZero}_S[i]$ và $\texttt{countZero}_T[i]$ lần lượt là số lượng ký tự `0` trong tiền tố độ dài i ở xâu S và T, $\texttt{pos}[i]$ là vị trí tương ứng của ký tự thứ i của xâu S ở xâu T.

Do số lượng ký tự `1` nằm bên trái ký tự `1` thứ i trong cả xâu S và T là như nhau nên ta có thể rút gọn công thức như về sau của công thức trên.

Subtask 2: Vét cạn

Duyệt hết các xâu t có cùng số lượng ký tự `1` với xâu s ban đầu và dùng công thức trên để tính số lượng thao tác tối thiểu. Nếu số thao tác $\le k$ thì t là một xâu hợp lệ, ta cập nhật đáp án theo định nghĩa đề bài.

- Độ phức tạp: $O(2^n \cdot n)$.

Subtask 3: Quy hoạch động

Từ công thức tính như trên, ta sẽ sử dụng Quy hoạch động để quản lý đóng góp của từng ký tự `1` vào tổng số thao tác (Contribution to the Sum).

Ta gọi `dp[diff][one][zero][val]` là số thao tác để biến đoạn `one` + `zero` kí tự đầu tiên (gồm `one` kí tự `1`, `zero` kí tự `0`) thành đoạn có `diff` cặp liền kề khác nhau và giá trị cuối cùng là `val` (với `val` = 1 hoặc `val` = 0).

Ý tưởng chung là ta sẽ tính giá trị của `dp`[...] với mọi `diff` từ 1 đến n rồi lấy `diff` lớn nhất sao cho $\min(dp[\texttt{diff}][...][0], dp[\texttt{diff}][...][1]) \le k$.

Để tính `dp[diff][one][zero][val]`, ta thấy:

- Nếu `val` = 0, tức là ký tự cuối trong xâu đang xét là 0, ký tự này sẽ không đóng góp vào tổng và ta truy hồi về trạng thái có `one` ký tự `1` và `zero` − 1 ký tự `0`.
- Nếu `val` = 1, tức là ký tự cuối trong xâu đang xét là 1. Để tính đóng góp của ký tự này, ta tính giá trị p là vị trí của ký tự `1` thứ `one` trong xâu s. Ta biết được rằng ký tự `1` này sẽ được đặt ở vị trí `one` + `zero` trong xâu mới nên đóng góp của nó là $|p - (\texttt{one} + \texttt{zero})|$. Sau đó, ta truy hồi về trạng thái có `one` − 1 ký tự `1` và `zero` ký tự `0`.

Việc xây dựng công thức Quy hoạch động cụ thể sẽ được nhường lại cho độc giả.

- Độ phức tạp bộ nhớ: $O(n^3)$.
- Độ phức tạp thời gian: $O(n^3)$.

Cài đặt (tham khảo)

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
typedef long double ld;
typedef pair<ll,ll> pl;
```

```cpp
typedef pair<int,int> pii;
typedef tuple<int,int,int> tt;

#define all(a) a.begin(), a.end()
#define filter(a) a.erase(unique(all(a)), a.end())

int dp[303][303][303][2], onePos[303];

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n, k; cin >> n >> k;
    string s; cin >> s;
    s = " " + s;

    /// pre-calculations
    int zeroCount = 0, oneCount = 0;
    for (int i = 1; i <= n; i++) {
        if (s[i] == '1') onePos[++oneCount] = i;
        else zeroCount++;
    }

    /// setup base-cases
    for (int diff = 0; diff <= n; diff++)
        for (int i = 0; i <= zeroCount; i++)
            for (int j = 0; j <= oneCount; j++)
                for (int last = 0; last < 2; last++) dp[diff][i][j][last] = INT_MAX;
    dp[0][0][0][0] = dp[0][0][0][1] = 0;

    /// run DP
    for (int diff = 1; diff <= n; diff++) {
        for (int zCur = 0; zCur <= zeroCount; zCur++) {
            for (int oCur = 0; oCur <= oneCount; oCur++) {
                if (!zCur && !oCur) continue;
                // calculate dp[diff][zCur][oCur][1]
                if (oCur) {
                    int oldPos = onePos[oCur], newPos = zCur + oCur, cost = abs(oldPos - newPos);
                    if (diff) { // place a '0' before it
                        int recur = dp[diff - 1][zCur][oCur - 1][0];
                        if (recur != INT_MAX)
                            dp[diff][zCur][oCur][1] = min(dp[diff][zCur][oCur][1], recur + cost);
                    }
                    if (true) { // place a '1' before it
                        int recur = dp[diff][zCur][oCur - 1][1];
                        if (recur != INT_MAX)
                            dp[diff][zCur][oCur][1] = min(dp[diff][zCur][oCur][1], recur + cost);
                    }
                }

                // calculate dp[diff][zCur][oCur][0]
                if (zCur) {
                    if (diff) { // place a '1' before it
                        dp[diff][zCur][oCur][0] = min(dp[diff][zCur][oCur][0], dp[diff - 1][zCur - 1][oCur][1]);
                    }
                    if (true) { // place a '0' before it
                        dp[diff][zCur][oCur][0] = min(dp[diff][zCur][oCur][0], dp[diff][zCur - 1][oCur][0]);
                    }
                }
            }
        }
```

```
    }

    int ans = 0;
    for (int diff = 2; diff <= n; diff++) {
        if (min(dp[diff][zeroCount][oneCount][0], dp[diff][zeroCount][oneCount][1]) <= k) ans =
     diff - 1;
    }
    cout << ans;

    return 0;
}
```

— HẾT —