

Giáo án TechSciX Camp 2023 môn tin học

Buổi 1. Số nguyên tố và các vấn đề liên quan.

I. Số nguyên tố và Hợp số

1. Giới thiệu

Số nguyên tố là số nguyên dương lớn hơn 1 và chỉ có đúng hai ước là 1 và chính nó.

Hợp số, là các số nguyên dương lớn hơn 1 và có nhiều hơn hai ước.

Lấy ví dụ: 5 là một số nguyên tố, vì nó chỉ có đúng hai ước là 1 và 5. Ngược lại 10 là một hợp số vì nó có bốn ước là 1, 2, 5 và 10. Số nguyên tố và các vấn đề xoay quanh nó luôn là một chủ đề được yêu thích trong Toán học nói chung và lập trình thi đấu nói riêng.

2. Kiểm tra tính nguyên tố của một số

2.1. Giải thuật cơ sở

Ý tưởng ban đầu rất đơn giản: Ta duyệt qua tất cả các số nguyên từ 2 tới $n - 1$, nếu như có số nào là ước của N thì kết luận n không phải số nguyên tố. Giải thuật có độ phức tạp $O(n)$.

```
bool is_prime(int n)
{
    if (n < 2)
        return false;

    for (int i = 2; i < n; ++i)
        if (N % i == 0)
            return false;

    return true;
}
```

2.2. Cải tiến:

Xuất phát từ nhận xét sau: Giả sử số nguyên dương n có ước là d ($0 < d \leq \sqrt{n}$), khi đó n sẽ có thêm một ước là $\frac{n}{d}$ ($\sqrt{n} \leq \frac{n}{d} \leq n$). Như vậy ta chỉ cần kiểm tra các số nguyên từ 2 tới \sqrt{n} xem n có chia hết cho số nào không, nếu không thì kết luận n là số nguyên tố. Giải thuật có độ phức tạp chỉ là $O(\sqrt{n})$.

```
bool is_prime(int n)
{
    if (n < 2)
        return false;

    for (int i = 2; i * i <= n; ++i)
        if (n % i == 0)
            return false;

    return true;
}
```

II. Sàng lọc số nguyên tố Eratosthenes

1. Giới thiệu

Sàng Eratosthenes là một giải thuật cổ xưa do nhà Toán học người Hy Lạp Eratosthenes phát minh ra để tìm các số nguyên tố nhỏ hơn 100. Tương truyền, khi tìm ra thuật toán, ông đã lấy lá cọ và ghi tất cả các số từ 2 cho đến 100 lên đó, sau đó chọc thủng các hợp số và giữ nguyên các số nguyên tố. Bảng số nguyên tố còn lại trông rất giống một cái sàng. Do đó, nó có tên là sàng Eratosthenes.

	2	3	4	5	6	7	8	9	10	Primzahlen; (số nguyên tố)
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	49	50	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	

Với sự phát triển của máy tính, sàng Eratosthenes trở thành một công cụ rất hữu dụng để tìm ra các số nguyên tố trong một khoảng nhất định, với điều kiện bộ nhớ có thể lưu trữ được.

2. Cài đặt giải thuật

Nguyên lý hoạt động của sàng Eratosthenes như sau: Xét các số nguyên tố từ 2 tới \sqrt{n} , với mỗi số nguyên tố ta sẽ đánh dấu các bội của nó mà lớn hơn nó đều là hợp số. Sau khi duyệt xong, tất cả các số chưa được đánh dấu sẽ là số nguyên tố. Dưới đây cài đặt sàng lọc các số nguyên tố từ 1 tới n .

```

bool is_prime[n + 1];

void eratosthenes_sieve(int n)
{
    // Mảng đánh dấu một số có phải số nguyên tố không.
    // Ban đầu giả sử mọi số đều là nguyên tố.
    for (int i = 0; i <= n; ++i)
        is_prime[i] = true;
    // 0 và 1 không phải số nguyên tố.
    is_prime[0] = is_prime[1] = false;

    for (int i = 2; i * i <= n; ++i)
        if (is_prime[i]) // Nếu i là số nguyên tố
            for (int j = i * i; j <= n; j += i) // Loại bỏ các bội của i lớn hơn
                is_prime[j] = false;
}

```

Bạn đọc có thể thắc mắc tại sao các bội của i lại không bắt đầu từ $2.i$. Lý do là vì, vòng lặp duyệt các số nguyên tố tăng dần, khi tới số nguyên tố i thì các bội $2.i, 3.i, \dots, (i - 1).i$ đều đã bị loại đi trước đó bởi các số nguyên tố nhỏ hơn i rồi. Cũng chính nhờ điều này nên vòng lặp bên ngoài chỉ cần duyệt từ 2 tới \sqrt{n} , giúp giảm độ phức tạp của giải thuật đi nhiều.

Đánh giá độ phức tạp

Quan sát cài đặt, ta nhận thấy:

- Với $i = 2$, vòng lặp bên trong lặp $\frac{n}{2}$ lần.
 - Với $i = 3$, vòng lặp bên trong lặp $\frac{n}{3}$ lần.
 - Với $i = 5$, vòng lặp bên trong lặp $\frac{n}{5}$ lần.
- ...

Tổng số lần lặp sẽ là $n \cdot (\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \dots)$, độ phức tạp sẽ tiến tới $O(n \cdot \log n)$.

3. Sàng số nguyên tố trên đoạn

Ở một số trường hợp, người ta cần tìm các số nguyên tố trên đoạn $[L, R]$ cho trước và R có thể lên tới 10^{12} , với điều kiện có thể tạo được một mảng có độ dài $(R - L + 1)$.

Ý tưởng giải thuật như sau: Sàng lọc trước một mảng gồm các số nguyên tố trong đoạn $[2 \dots \sqrt{R}]$, sau đó duyệt qua các số nguyên tố này, loại bỏ các bội của chúng nằm trong đoạn $[L, R]$. Code dưới đây cải tiến một chút để bỏ bớt bước tạo mảng số nguyên tố trong đoạn $[2 \dots \sqrt{R}]$, nhằm tiết kiệm thời gian chạy.

```
vector < bool > void range_eratosthenes(int L, int R)
{
    int maxn = sqrt(R);
    vector < bool >
    int range = R - L + 1;
    for (int i = 2; i <= range; ++i)
        is_prime[i] = true;

    // Duyệt các bội của i từ bội nhỏ nhất thuộc đoạn [L, R].
    for (long long i = 2; i * i <= R; ++i)
        if (is_prime[i])
            for (long long j = max(i * i, (L + i - 1) / i * i); j <= R; j += i)
                is_prime[j - L] = false;
```

```
if (L == 1)
    is_prime[0] = false;
}
```

Như vậy với một số X trong đoạn $[L, R]$, X là số nguyên tố khi và chỉ khi $\text{is_prime}[X - L] = \text{true}$. Thuật toán có độ phức tạp là $O((R - L + 1).\log(R) + \sqrt{R})$. Trên thực tế nó chạy khá nhanh.

III. Phân tích thừa số nguyên tố

Vấn đề phân tích thừa số nguyên tố cũng khá được quan tâm trong lập trình thi đấu, và nó còn có một số ứng dụng khác trong số học. Dưới đây chúng ta sẽ xem xét vài phương pháp phân tích thừa số nguyên tố thường dùng.

1. Giải thuật cơ sở

Ta xét mọi số nguyên tố bắt đầu từ 2, nếu n chia hết cho số nguyên tố p thì chia n cho p tới khi không thể chia hết nữa, rồi tăng p lên và lặp lại công việc tới khi $n = 1$. Trên thực tế thừa số nguyên tố chính là thành phần cấu tạo nên một ước của n , do đó khi tách hết một thừa số nguyên tố x khỏi n thì n sẽ không thể chia hết cho các bội lớn hơn x nữa.

Cài đặt

```
vector < int > extract(int n)
{
    int p = 2;
    vector < int > prime_factor; // Lưu thừa số nguyên tố vào vector.

    while (n > 1)
    {
        while (N % p == 0)
        {
            prime_factor.push_back(p);
            n /= p;
        }

        ++p;
    }
}
```

```
    return prime_factor;
}
```

2. Cải tiến lần 1

Xuất phát từ nhận xét sau: Không thể xảy ra trường hợp mọi thừa số nguyên tố của n đều lớn hơn \sqrt{n} , do đó chúng ta chỉ cần xét các ước của n từ 2 tới \sqrt{n} và chia dần n cho các ước của nó tới khi n bằng 1. Nếu không thể tìm được ước nào từ 2 tới \sqrt{n} thì n phải là một số nguyên tố. Độ phức tạp giải thuật là $O(\sqrt{n})$.

Cài đặt

```
vector < int > extract(int n)
{
    vector < int > prime_factor;

    for (int i = 2; i * i <= n; ++i)
        while (n % i == 0)
    {
        prime_factor.push_back(i);
        n /= i;
    }

    if (n > 1)
        prime_factor.push_back(i);

    return prime_factor;
}
```

3. Phân tích thừa số nguyên tố bằng sàng Eratosthenes

Tùy hai giải thuật trên ta thấy: Ở mỗi bước phân tích cần tìm ra ước nguyên tố nhỏ nhất của n rồi chia n cho ước đó. Ta sẽ thay đổi sàng Eratosthenes đi một chút để lấy được ngay ước nguyên tố nhỏ nhất của n trong $O(1)$ ở mỗi bước phân tích, điều này sẽ giúp giảm thời gian chạy đi đáng kể.

Cài đặt

```
int smallest_divisor[max_value + 1];
bool is_prime[max_value + 1];
```

```

void eratosthenes_sieve(int max_value)
{
    // Mảng lưu ước nguyên tố nhỏ nhất của các số trong đoạn [1, max_value].
    fill(smallest_divisor + 1, smallest_divisor + max_value + 1, 0);
    fill(is_prime, is_prime + max_value + 1, true);
    is_prime[0] = is_prime[1] = false;

    for (int i = 2; i * i <= max_value; ++i)
        if (is_prime[i]) // Nếu i là số nguyên tố
            for (int j = i * i; j <= max_value; j += i)
            {
                is_prime[j] = false;

                // Uớc nguyên tố nhỏ nhất của j là i.
                if (smallest_divisor[j] == 0)
                    smallest_divisor[j] = i;
            }

    // Xét riêng các trường hợp i là số nguyên tố,
    // ước nguyên tố nhỏ nhất là chính nó.
    for (int i = 2; i <= max_value; ++i)
        if (is_prime[i])
            smallest_divisor[i] = i;
}

vector < int > extract(int n)
{
    // Sàng số nguyên tố tới một giá trị max_value nào đó.
    eratosthenes_sieve(max_value);

    vector < int > prime_factor;

    while (n > 1)
    {
        int p = smallest_divisor[n];
        prime_factor.push_back(p);
        n /= p;
    }

    return prime_factor;
}

```

Mặc dù việc thực hiện sàng Eratosthenes vẫn mất $O(n \log n)$, tuy nhiên thao tác phân tích một số p thành thừa số nguyên tố chỉ mất độ phức tạp $O(\log p)$. Điều này sẽ rất có lợi trong

các bài toán phải phân tích thừa số nguyên tố nhiều lần.

4. Đếm số ước của một số nguyên

Giả sử ta phân tích được n thành các thừa số nguyên tố ở dạng:

$$n = p_1^{k_1} \times p_2^{k_2} \times \dots \times p_m^{k_m}$$

Các ước số của n sẽ phải có dạng:

$$p_1^{r_1} \times p_2^{r_2} \times \dots \times p_m^{r_m}$$

với $0 \leq r_1 \leq k_1, 0 \leq r_2 \leq k_2, \dots, 0 \leq r_m \leq k_m$.

Theo nguyên lý nhân, ta thấy: r_1 có $k_1 + 1$ cách chọn, r_2 có $k_2 + 2$ cách chọn,..., r_m có $k_m + 1$ cách chọn. Như vậy số lượng ước của n sẽ được tính theo công thức:

$$F_n = (k_1 + 1) \times (k_2 + 1) \times \dots \times (k_m + 1)$$

Từ đây, ta có ý tưởng đếm số ước của một số nguyên dương n như sau:

- Phân tích n thành thừa số nguyên tố.
- Đặt một biến `cnt_x` là số lần xuất hiện của thừa số nguyên tố x trong phân tích của n .
Ta vừa phân tích n vừa cập nhật số lượng thừa số nguyên tố lên biến `cnt_x`.

Cài đặt 1

Tính số lượng ước nguyên dương của số n bằng phân tích nguyên tố trong $O(\sqrt{n})$:

```
int count_divisors(int n)
{
    int total_divisor = 1; // Chắc chắn n có ước là 1.
    for (int i = 2; i * i <= n; ++i)
    {
        int cnt = 0; // Đếm số lượng thừa số nguyên tố i trong phân tích của N.
        while (n % i == 0)
        {
            ++cnt;
            N /= i;
        }

        total_divisors *= (cnt + 1);
    }
}
```

```

if (n > 1)
    total_divisors *= 2;

return total_divisors;
}

```

Trong trường hợp cần đếm ước của nhiều số (khoảng 10^6 số chẳng hạn), và mỗi số đều có giá trị nhỏ hơn hoặc bằng 10^7 , thì giải thuật trên sẽ bị Time Limit Exceeded. Khi đó, ta có thể cải tiến thuật toán bằng cách áp dụng luôn việc phân tích thừa số nguyên tố sử dụng Sàng Eratosthenes, khi đó việc đếm ước của mỗi số sẽ chỉ còn độ phức tạp là $O(\log n)$.

Cài đặt 2

Trong cài đặt này, ta sẽ tái sử dụng lại code Sàng số nguyên tố Eratosthenes và mảng `smallest_divisor` ở phần phân tích thừa số nguyên tố bằng Sàng Eratosthenes bên trên.

Tính số lượng ước nguyên dương của số n bằng phân tích nguyên tố trong $O(\log n)$:

```

int count_divisors(int n)
{
    int total_divisors = 1;

    while (n > 1)
    {
        int d = smallest_divisor[n], power = 0;
        while (n % d == 0)
        {
            ++power;
            n /= d;
        }

        total_divisors *= (power + 1);
    }

    return total_divisors;
}

```

5. Tính tổng các ước số nguyên dương của một số nguyên dương

Định lý

Nếu một số nguyên dương n khi phân tích ra thừa số nguyên tố có dạng:

$$n = p_1^{k_1} \times p_2^{k_2} \times \dots \times p_m^{k_m} \quad (k_i \neq 0; \forall i : 1 \leq i \leq m)$$

thì tổng các ước số nguyên dương của nó được tính theo công thức:

$$\sigma(n) = \prod_{i=1}^m \left(\frac{p_i^{k_i+1} - 1}{p_i - 1} \right)$$

Chứng minh: Các ước số của n sẽ phải có dạng:

$$p_1^{r_1} \times p_2^{r_2} \times \dots \times p_m^{r_m}$$

với $0 \leq r_1 \leq k_1, 0 \leq r_2 \leq k_2, \dots, 0 \leq r_m \leq k_m$.

→ Tổng các ước của n là:

$$\begin{aligned} \sigma(N) &= \sum_{r_1=0}^{k_1} \sum_{r_2=0}^{k_2} \cdots \sum_{r_m=0}^{k_m} (p_1^{r_1} \times p_2^{r_2} \times \cdots \times p_m^{r_m}) \\ &= \sum_{r_1=0}^{k_1} p_1^{r_1} \times \sum_{r_2=0}^{k_2} p_2^{r_2} \times \cdots \times \sum_{r_m=0}^{k_m} p_m^{r_m} \quad (1) \end{aligned}$$

Mà ta có công thức dãy cấp số nhân sau:

$$p^0 + p^1 + p^2 + \cdots + p^n = \frac{p^{n+1} - 1}{p - 1} \quad (2)$$

Từ (1) và (2), ta có:

$$\sigma(n) = \frac{p_1^{k_1+1} - 1}{p_1 - 1} \times \frac{p_2^{k_2+1} - 1}{p_2 - 1} \times \cdots \times \frac{p_m^{k_m+1} - 1}{p_m - 1}$$

Cài đặt 1

Để tránh tràn số, ở đây tôi đặt luôn tất cả các số mang kiểu dữ liệu long long. Tùy vào bài toán mà các bạn sẽ điều chỉnh lại cho phù hợp.

Tuy nhiên, giải thuật này có sử dụng tới kỹ thuật tính nhanh lũy thừa a^b trong $O(\log b)$ bằng giải thuật Bình phương và Nhân. Các bạn có thể đọc trước về giải thuật này tại [đây](#).

Tính tổng các ước của một số nguyên dương n trong $O(\sqrt{n} \times \log n)$:

```

long long exponentiation(long long A, long long B)
{
    if (B == 0)
        return 1LL;

    long long half = exponentiation(A, B / 2LL);

    if (B & 1)
        return half * half * A;
    else
        return half * half;
}

long long get_sum_divisors(long long n)
{
    if (n == 1)
        return 1;

    long long x = 1, y = 1;
    for (int i = 2; i * i <= n; ++i)
    {
        long long cnt = 0; // Đếm số lượng thừa số nguyên tố i trong phân tích c
        while (n % i == 0)
        {
            ++cnt;
            n /= i;
        }

        if (cnt != 0)
        {
            x *= (exponentiation(i, cnt + 1) - 1);
            y *= (i - 1);
        }
    }

    if (n > 1)
        x *= n * n - 1, y *= (n - 1);

    return x / y;
}

```

Cũng tương tự như khi đếm ước, ta có thể sử dụng sàng Eratosthenes để tính tổng các ước của một số nguyên dương n , phương pháp này sẽ rất hữu dụng trong các bài toán cần tính nhanh tổng các ước của n .

Cài đặt 2

Trong cài đặt này, ta sẽ tái sử dụng lại code Sàng số nguyên tố Eratosthenes và mảng smallest_divisor ở phần phân tích thừa số nguyên tố bằng Sàng Eratosthenes bên trên.

Tuy nhiên, do dùng tới Sàng nguyên tố nên giải thuật này chỉ có thể phân tích các số có giá trị khoảng 10^7 trở xuống. Các bạn cần lựa chọn thuật toán phù hợp với ràng buộc đề bài.

Tính tổng các ước nguyên dương của một số n trong $O(\log n)$:

```
long long exponentiation(long long A, long long B)
{
    if (B == 0)
        return 1LL;

    long long half = exponentiation(A, B / 2LL);

    if (B & 1)
        return half * half * A;
    else
        return half * half;
}

long long get_sum_divisors(int n)
{
    long long x = 1, y = 1;

    while (n > 1)
    {
        int d = smallest_divisor[n], power = 0;
        while (n % d == 0)
        {
            ++power;
            n /= d;
        }

        x *= (exponentiation(d, power + 1) - 1);
        y *= (d - 1);
    }

    return x / y;
}
```

IV. Bài tập minh họa

1. Số nguyên tố đặc biệt

Đề bài

Một số nguyên dương X được gọi là số nguyên tố đặc biệt khi và chỉ khi nó có đúng 3 ước nguyên dương phân biệt.

Yêu cầu: Cho trước số nguyên dương N . Hãy đếm số lượng số nguyên tố đặc biệt từ 1 tới N ?

Input:

- Chứa duy nhất số nguyên dương N .

Ràng buộc:

- $1 \leq N \leq 10^9$.

Subtasks

- Subtask 1 (50% số điểm): $1 \leq N \leq 10^3$.
- Subtask 2 (50% số điểm): Không có ràng buộc gì thêm.

Output:

- In ra danh sách các số nguyên tố đặc biệt từ 1 tới N , mỗi số trên một dòng.

Sample Input:

5

Sample Output:

4

Ý tưởng

Subtask 1

Duyệt tất cả các số và đếm ước của chúng.

Độ phức tạp: $O(n \times \sqrt{n})$.

Subtask 2

Nhận thấy, số có 3 ước nguyên dương phân biệt chắc chắn phải là bình phương của một số nguyên tố. Vậy ta sẽ đi tìm số lượng số x là số nguyên tố thỏa mãn: $1 \leq x \leq \sqrt{n}$, khi đó x^2 chính là một số nguyên tố đặc biệt.

Độ phức tạp: $O(\sqrt{n})$.

Cài đặt

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

using namespace std;

bool is_prime(int N)
{
    if (N < 2) return false;

    for (int i = 2; i * i <= N; ++i)
        if (N % i == 0)
            return false;

    return true;
}

void solution(int N)
{
    for (int i = 1; i * i <= N; ++i)
        if (is_prime(i))
            cout << i * i << endl;
}

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    int N;
    cin >> N;
```

```

solution(N);

return 0;
}

```

2. Tìm số mũ

Đề bài

Cho trước hai số nguyên N và P .

Yêu cầu: Xác định giá trị M nguyên dương lớn nhất để P^M là ước số của $N!$

Input:

- Một dòng duy nhất chứa hai số nguyên N và P .

Ràng buộc:

- $1 \leq N, P \leq 30000$.

Output:

- Đưa ra số M tìm được. Dữ liệu đảm bảo luôn tồn tại giá trị M nguyên dương thỏa mãn yêu cầu.

Sample Input:

7 3

Sample Output:

2

Ý tưởng

Giả sử $n!$ có phân tích nguyên tố là:

$$p_1^{k_1} \times p_2^{k_2} \times \cdots \times p_x^{k_x}$$

Thì các ước số của $n!$ phải có dạng:

$$p_1^{k'_1} \times p_2^{k'_2} \times \cdots \times p_x^{k'_x})$$

với $k'_1 \leq k_1, k'_2 \leq k_2, \dots, k'_x \leq k_x$.

Suy ra để P^m là ước của $n!$ thì P^m cũng phải có dạng như trên. Giả sử phân tích nguyên tố của P là:

$$q_1^{r_1} \times q_2^{r_2} \times \cdots \times q_l^{r_l}$$

Thì P^m có phân tích là:

$$q_1^{r_1.m} \times q_2^{r_2.m} \times \cdots \times q_l^{r_l.m}$$

Nếu trong phân tích của P^m tồn tại một số nguyên tố không xuất hiện trong phân tích của $n!$ thì chắc chắn không tìm được giá trị m . Tuy nhiên đề bài nói rằng chắc chắn luôn luôn tìm được m . Do đó, phân tích nguyên tố của P^m sẽ chỉ bao gồm các thừa số nguyên tố trong phân tích của $n!$:

$$p_1^{r_1.m} \times p_2^{r_2.m} \times \cdots \times p_x^{r_x.m}$$

Vì P^m là ước của $n!$ nên:

$$\begin{aligned} r_1.m &\leq k_1; r_2.m \leq k_2; \dots; r_x.m \leq k_x \\ \Leftrightarrow m &\leq \left\lfloor \frac{k_1}{r_1} \right\rfloor; m \leq \left\lfloor \frac{k_2}{r_2} \right\rfloor; \dots; m \leq \left\lfloor \frac{k_x}{r_x} \right\rfloor \\ \Leftrightarrow m \max &= \min \left(\left\lfloor \frac{k_i}{r_i} \right\rfloor \right); \forall i : 1 \leq i \leq x \end{aligned}$$

Vì $n, p \leq 30000$ nên chỉ cần áp dụng phân tích thừa số nguyên tố trong $O(\sqrt{n})$ là có thể vượt qua ràng buộc thời gian 1s.

Cài đặt

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

using namespace std;

void extract(int n, vector < int > &power)
{
    for (int i = 2; i * i <= n; i++)
    {
        while (n % i == 0)
```

```

    {
        n /= i;
        power[i]++;
    }
}

if (n > 1)
    power[n]++;
}

void solution(int n, int p)
{
    vector < int > power1(n + 1);
    for (int i = 1; i <= n; i++)
        extract(i, power1);

    vector < int > power2(n + 1);
    extract(p, power2);

    int res = INT_MAX;
    for (int i = 2; i <= n; i++)
        if (power1[i] != 0 && power2[i] != 0)
            res = min(res, power1[i] / power2[i]);
}

cout << res;
}

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, p;
    cin >> n >> p;

    solution(n, p);

    return 0;
}

```

3. Quà sinh nhật

Đề bài

Nguyên nhận được món quà sinh nhật thú vị. Đó là một dãy số nguyên dương. Anh ta quyết định biến đổi dãy số nhận được thành dãy toàn số 1.

Tại mỗi bước, anh ta chọn ra một tập hợp các số có cùng một ước số là số nguyên tố p nào đó, chia tất cả các số trong tập hợp này cho số p .

Yêu cầu: Hỏi rằng Nguyên phải thực hiện tối thiểu bao nhiêu bước biến đổi để thu được dãy toàn số 1?

Input:

- Dòng đầu tiên ghi số nguyên dương T là số bộ dữ liệu. Tiếp theo là T nhóm dòng, mỗi nhóm dòng mô tả một bộ dữ liệu theo qui cách sau:
 - Dòng đầu tiên ghi số nguyên dương N - số lượng số của dãy số.
 - Dòng thứ hai chứa N số nguyên a_1, a_2, \dots, a_N phân tách nhau bởi dấu cách - dãy số được cho.

Ràng buộc:

- $1 \leq T \leq 5$.
- $1 \leq N \leq 10^5$.
- $1 \leq a_i \leq 10^6; \forall i : 1 \leq i \leq N$.

Output:

- Với mỗi bộ dữ liệu in ra trên một dòng số nguyên - số lượng tối thiểu các bước cần thực hiện để thu được dãy chỉ chứa toàn số 1.

Sample Input:

```
1
3
1 2 4
```

Sample Output:

```
2
```

Ý tưởng

Mỗi lần ta chọn một số nguyên tố p và chia mọi số trong mảng A có chứa p cho nó, vì vậy ta có thể hiểu ý tưởng của bài là xét mọi thừa số nguyên tố xuất hiện trong phân tích nguyên tố

của các a_i , và lần lượt đếm số bước cần thực hiện để loại bỏ mỗi thừa số đó ra khỏi mọi số trong mảng A .

Ta sẽ phân tích ước nguyên tố ra với mỗi số a_i trong $O(\log(a_i))$ sử dụng sàng số nguyên tố. Với mỗi thừa số nguyên tố p , giả sử nó xuất hiện trong k số của mảng A là $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ với số lần xuất hiện lần lượt là c_1, c_2, \dots, c_k ; thì tổng số lần chia cần thiết để loại bỏ p khỏi toàn bộ k số đó sẽ là $\max(c_1, c_2, \dots, c_k)$ lần, bởi vì mỗi khi ta chia một số a_i cho p thì các số khác cũng chứa p cũng sẽ được gom nhóm cùng a_i để chia một lần.

Vậy dùng một mảng $\text{max_power}[p]$ để lưu số mũ cao nhất của thừa số nguyên tố p trong phân tích nguyên tố của tất cả các a_i . Sau đó, lưu một mảng primes chứa mọi số nguyên tố khác nhau trong quá trình phân tích các a_i . Kết quả cuối cùng sẽ là:

$$\sum \text{max_power}[p]; \forall p \in \text{primes}$$

Độ phức tạp: $(T \times n \cdot \log(n))$.

Cài đặt

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

using namespace std;

const int maxn = 1e5, max_value = 1e6;
int a[maxn + 1], smallest_divisor[max_value + 1];

void eratosthenes_sieve(int max_value)
{
    vector < bool > is_prime(max_value + 1, true);
    is_prime[0] = is_prime[1] = false;

    for (int i = 2; i * i <= max_value; ++i)
        if (is_prime[i])
            for (int j = i * i; j <= max_value; j += i)
            {
                is_prime[j] = false;

                if (smallest_divisor[j] == 0)
                    smallest_divisor[j] = i;
            }

    for (int i = 2; i <= max_value; ++i)
        if (is_prime[i])
```

```

        smallest_divisor[i] = i;
    }

void solution(int n, int a[])
{
    vector < int > max_power(max_value + 1, 0);
    vector < int > appear(max_value + 1, false);
    vector < int > primes;

    for (int i = 1; i <= n; ++i)
    {
        int x = a[i];
        while (x > 1)
        {
            int p = smallest_divisor[x], power = 0;
            while (x % p == 0)
            {
                ++power;
                x /= p;
            }

            max_power[p] = max(max_power[p], power);
            if (!appear[p])
                appear[p] = true, primes.push_back(p);
        }
    }

    int res = 0;
    for (int p: primes)
        res += max_power[p];

    cout << res << endl;
}

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

    eratosthenes_sieve(max_value);

    int t;
    cin >> t;

    while (t--)
    {

```

```

int n;
cin >> n;

for (int i = 1; i <= n; ++i)
    cin >> a[i];

solution(n, a);
}

return 0;
}

```

V. Tài liệu tham khảo

- <https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html>
- https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
- <https://vnoi.info/wiki/translate/he/Number-Theory-2.md>
- <https://cp-algorithms.com/algebra/factorization.html>
- <https://www.geeksforgeeks.org/segmented-sieve-print-primes-in-a-range/?ref=rp>
- <https://www.mathvn.com/2020/01/chung-minh-ing-li-ve-ham-tongcac-uoc.html>
- <https://www.mathvn.com/2020/01/cong-thuc-tinh-tongcac-uoc-nguyen.html>

Buổi 2. Phép nhân Ấn Độ - Thuật toán bình phương và nhân

Trong chuyên đề này, chúng ta sẽ cùng nghiên cứu về hai kĩ thuật khá quen thuộc và có tính ứng dụng cao trong các bài toán số học, đó là **Phép nhân Ấn Độ** và **Thuật toán bình phương và nhân** - những kĩ thuật sẽ giúp các bạn tính toán $a \times n$ và a^n trong thời gian $O(\log(n))$. Mặc dù nghe có vẻ khá vô dụng (bởi vì thực ra phép nhân chỉ cần thực hiện trực tiếp), nhưng đối với một số trường hợp thì chúng lại rất có ích (đặc biệt với ngôn ngữ C++).

Trước khi nghiên cứu chuyên đề này, các bạn cần phải có kiến thức về **Giải thuật đệ quy**, bởi vì các cài đặt đều sẽ sử dụng đệ quy. Nếu chưa nắm được thì các bạn hãy vào đọc về nó tại [đây](#).

I. Phép nhân Ấn Độ

1. Đặt vấn đề

Xét một bài toán đơn giản như sau: Cho hai số a, b ($a, b \leq 10^{18}$). Tính giá trị biểu thức $(a \times b) \% 10^9$.

Bài toán trên có thể dễ dàng giải quyết bằng tính chất phân phối của phép nhân đối với phép đồng dư thức: $(a \times b) \% 10^9 = [(a \% 10^9) \times (b \% 10^9)] \% 10^9$. Tuy nhiên, nếu như ta cần lấy số dư cho 10^{18} thì sao? Phép toán bằng tính chất phân phối bây giờ sẽ không thể thực hiện được, vì $[(a \% 10^{18}) \times (b \% 10^{18})] \leq 10^{36}$, dẫn đến kết quả của bước này sẽ bị vượt quá khả năng biểu diễn của kiểu số nguyên 64 bit trong C++.

Phép nhân Ấn Độ sử dụng để tính $(a \times b) \% M$ trong trường hợp tính chất phân phối với phép đồng dư thức không thể áp dụng được vì lí do tràn số. Tuy nhiên điều này chỉ xảy ra với C++, còn đối với Python thì sẽ không ảnh hưởng gì cả.

2. Phép nhân Ấn Độ với đồng dư thức

Nguyên lý phép nhân Ấn Độ rất đơn giản như sau:

$$(a \times b) = \begin{cases} a \times \frac{b}{2} + a \times \frac{b}{2}, & \text{nếu } b \text{ là số chẵn .} \\ a \times \frac{b}{2} + a \times \frac{b}{2} + a, & \text{nếu } b \text{ là số lẻ.} \end{cases}$$

Dựa trên lý thuyết này, ta sẽ kết hợp phép nhân Ấn Độ với tính chất phân phối của phép nhân, phép cộng với phép đồng dư thức để tính được $(a \times b) \% M$, với $M \leq 10^{18}$ mà không bị tràn số.

Lưu ý tính chất phân phối của phép cộng đối với phép đồng dư thức, tính chất này có thể áp dụng với cả phép nhân và phép trừ:

$$(a + b) \% M = [(a \% M) + (b \% M)] \% M$$

Cài đặt

Ngôn ngữ C++:

```
// Tính a * b % M
long long multiply_modulo(long long a, long long b, long long M)
{
    if (b == 0)
        return 0;

    long long t = multiply_modulo(a, b / 2, M) % M;
```

```

if (b & 1)
    return ((t + t) % M + a % M) % M;
else
    return (t + t) % M;
}

```

Ngôn ngữ Python:

```

## Tính a * b % M
def multiply_modulo(a, b, M):
    if b == 0:
        return 0

    t = multiply_modulo(a, b // 2, M)

    if b & 1:
        return ((t + t) % M + a % M) % M
    else:
        return (t + t) % M

```

Đánh giá độ phức tạp

Trong thuật toán, b liên tục giảm đi một nửa, nên độ phức tạp của giải thuật là $O(\log_2(b))$.

I. Thuật toán bình phương và nhân

1. Giải thuật chia để trị

Thông thường, để tính lũy thừa a^b , ta sẽ cần sử dụng b phép nhân liên tiếp trong một vòng lặp. Nếu như b là một số lớn cỡ 10^9 trở lên, thì việc tính toán sẽ mất rất nhiều thời gian, không thể đáp ứng được yêu cầu trong các bài toán lập trình. Mặt khác, kết quả của phép lũy thừa thường rất lớn, nên đề bài sẽ yêu cầu thí sinh in ra kết quả sau khi chia lấy dư cho một giá trị M nào đó.

Dựa trên tư tưởng phép nhân Ấn Độ, ta có thể điều chỉnh công thức một chút để tính được lũy thừa $a^b \% M$, với $a, b, M \leq 10^9$. Công thức đơn giản như sau:

$$(a \times b) = \begin{cases} a^{\frac{b}{2}} \times a^{\frac{b}{2}}, & \text{nếu } b \text{ là số chẵn .} \\ a^{\frac{b}{2}} \times a^{\frac{b}{2}} \times a, & \text{nếu } b \text{ là số lẻ.} \end{cases}$$

Cài đặt

Trong cài đặt dưới đây, tác giả sẽ cài đặt mẫu cả phiên bản đệ quy và phiên bản khử đệ quy của thuật toán.

Ngôn ngữ C++:

```
// Tính a^b mod M.
long long power_modulo(long long a, long long b, long long M)
{
    if (b == 0)
        return 1LL;

    long long half = power_modulo(a, b / 2, M) % M;

    if (b & 1)
        return (((half * half) % M) * (a % M)) % M;
    else
        return (half * half) % M;
}
```

Ngôn ngữ Python:

```
## Tính a^b % M
def power_modulo(a, b, M):
    if b == 0:
        return 1

    half = power_modulo(a, b // 2, M) % M

    if b & 1:
        return (half * half * a) % M
    else:
        return (half * half) % M
```

Ngoài ra, ta cũng có thể cài đặt giải thuật bằng phương pháp khử đệ quy để đẩy nhanh tốc độ thuật toán hơn một chút:

Ngôn ngữ C++:

```
// Tính a^b % M khử đệ quy.
long long power_modulo_non_recur(long long a, long long b, long long M)
{
    long long res = 1;
```

```

while (b)
{
    if (b & 1)
        res = (res * a) % M;

    a = (a * a) % M;
    b /= 2;
}

return res;
}

```

Ngôn ngữ Python:

```

def power_modulo_non_recur(a, b, M):
    res = 1
    while b != 0:
        if b % 1:
            res = (res * a) % M

        a = (a * a) % M
        b = b // 2

    return res

```

Đánh giá độ phức tạp

Trong thuật toán, b liên tục giảm đi một nửa, nên độ phức tạp của giải thuật là $O(\log_2(b))$. Mặc dù độ phức tạp của cả hai cách cài đặt là tương tự nhau, nhưng trong thực tế cách làm khử đệ quy sẽ chạy nhanh hơn một chút do không phải gọi đệ quy.

2. Tính $a^b \% M$ với $M \leq 10^{18}$

Trong trường hợp $M \leq 10^{18}$, dựa trên những gì đã phân tích ở phần I, phép nhân thông thường sẽ không thể áp dụng trong C++ vì lí do xảy ra tràn số. Vì vậy, ta sẽ kết hợp thêm phép nhân Ấn Độ trong trường hợp này. Độ phức tạp sẽ trở thành $O(\log_2(b)^2)$

Cài đặt

Ngôn ngữ C++:

```

long long power_modulo(long long a, long long b, long long M)
{
    if (b == 0)
        return 1LL;

    long long half = power_modulo(a, b / 2LL, M) % M;
    half = multiply_modulo(half, half, M);

    if (b & 1)
        return multiply_modulo(half, a, M);
    else
        return half;
}

```

Ngôn ngữ Python:

Mặc dù phép nhân trong Python không bị tràn số, tuy nhiên bản chất ngôn ngữ này vẫn sẽ phải sử dụng các thuật toán xử lý số nguyên lớn để thao tác khi dữ liệu quá to, nên có khả năng chương trình sẽ bị chạy quá thời gian. Vì vậy, tác giả vẫn đưa vào đoạn code bằng Python để các bạn sử dụng khi cần thiết:

```

def power_modulo(a, b, M):
{
    if b == 0:
        return 1;

    half = power_modulo(a, b // 2, M) % M;
    half = multiply_modulo(half, half, M);

    if (b & 1)
        return multiply_modulo(half, a, M);
    else
        return half;
}

```

Đánh giá độ phức tạp

Do sử dụng kết hợp cả thuật toán bình phương và nhân lẩn phép nhân Ấn Độ, nên độ phức tạp tổng quát sẽ là $O(\log^2 b)$.

3. Tính $a^b \% M$ trong trường hợp b là số lớn và M là số nguyên tố

Đối với các trường hợp b là số lớn - hiểu là các số nằm ngoài khả năng lưu trữ của kiểu số trong C++ và phải lưu bằng kiểu chuỗi - khi đó giải thuật tính $a^b \% M$ sẽ trở nên hơi phức tạp nếu như chúng ta cài đặt bằng các phép toán số lớn. Tuy nhiên, trong trường hợp M là một số nguyên tố, dựa vào một số tính chất số học, ta có thể thu gọn được việc tính toán như sau:

- Thứ nhất, cần biết định lý nhỏ Fermat được phát biểu như sau: Nếu M là một số nguyên tố thì:

$$a^{M-1} \equiv 1 \pmod{M}, \text{ với } a \% M \neq 0$$

- Lại có: $a^b \% M = (a^{M-1} \cdot a^{M-1} \dots a^{M-1} \cdot a^x) \% M$, với a^{M-1} lặp lại $\lfloor \frac{b}{M-1} \rfloor$ lần và $x = b \% (M-1)$. Từ đây suy ra:

$$a^b \% M = (1 \cdot 1 \cdot 1 \dots 1 \cdot a^x) \% M = a^x \% M$$

Tới đây chúng ta có thể áp dụng thuật toán bình phương và nhân một cách bình thường mà không sợ bị tràn số. Tất nhiên vẫn sẽ cần lưu ý về giới hạn của M để lựa chọn phép nhân thông thường hay phép nhân Ấn Độ.

Việc cài đặt xin dành lại cho bạn đọc.

III. Một số bài toán minh họa

1. Chữ số tận cùng

Đề bài

Cho hai số nguyên a và b . Cần tìm chữ số tận cùng của a^b ?

Input:

- Một dòng duy nhất chứa hai số nguyên a và b .

Ràng buộc:

- $1 \leq a, b \leq 10^9$.

Output:

- Đưa ra chữ số tận cùng của a^b .

Sample Input:

3 10

Sample Output:

9

Ý tưởng

Để lấy chữ số tận cùng của một số nguyên dương n , ta lấy n chia cho 10 và kết quả là số dư của phép chia.

Áp dụng công thức trên, ta có kết quả bài toán là $a^b \% 10$.

Cài đặt

Ngôn ngữ C++:

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

using namespace std;

void enter(long long &a, long long &b)
{
    cin >> a >> b;
}

long long power_modulo(long long a, long long b, long long mod)
{
    if (b == 0)
        return 1LL;

    long long half = power_modulo(a, b / 2, mod) % mod;

    if (b & 1)
        return (((half * half) % mod) * (a % mod)) % mod;
    else
        return (half * half) % mod;
}

main()
{
```

```

long long a, b;

enter(a, b);
cout << power_modulo(a, b, 10);

return 0;
}

```

Ngôn ngữ Python:

```

def power_modulo(a, b, mod):
    if b == 0:
        return 1

    half = power_modulo(a, b // 2, mod) % mod

    if b & 1:
        return (half * half * a) % mod
    else:
        return (half * half) % mod

if __name__ == '__main__':
    a = int(input())
    b = int(input())

    print(power_modulo(a, b, 10))

```

2. Đỗ xe

Đề bài

Một bãi đỗ xe có $2n - 2$ chỗ đỗ xe liên tiếp, và các CEO của bãi đỗ xe dự định lấp đầy bãi đỗ xe bằng 4 loại xe ở 4 màu khác nhau để làm lễ khai trương. Khi nhìn những chiếc xe xếp thành một đường thẳng, họ thấy rằng nếu như sắp xếp lại các xe sao cho có ít nhất một đoạn gồm đúng n chiếc xe cùng màu đứng cạnh nhau, thì bãi đỗ xe sẽ đẹp hơn.

Hãy giúp các CEO đếm xem có bao nhiêu cách sắp xếp như vậy? Biết rằng số lượng xe của mỗi loại đều lớn hơn số lượng chỗ đỗ xe.

Input:

- Chứa duy nhất số nguyên dương n .

Ràng buộc:

- $2 \leq n \leq 10^9$.

Output:

- In ra số nguyên duy nhất là số lượng cách sắp xếp những chiếc xe vào $2n - 2$ chỗ đỗ xe theo yêu cầu. Do kết quả có thể rất lớn, chỉ cần in ra số dư của nó sau khi chia cho $10^9 + 7$.

Sample Input:

3

Sample Output:

24

Ý tưởng

Ta phải chọn ra tổng cộng $2n - 2$ chiếc xe từ 4 loại xe khác nhau để đặt trên một đường thẳng, sao cho có chính xác n chiếc xe cùng màu đứng liên tiếp nhau. Những phương án chọn có thể là: Đặt n chiếc cùng màu ở n vị trí liên tiếp ở đầu đường thẳng, hoặc đặt ở n vị trí liên tiếp ở cuối đường thẳng, hoặc đặt ở n vị trí liên tiếp ở giữa đường thẳng.

Nếu như n chiếc cùng màu được đặt ở n vị trí đầu tiên hoặc n vị trí cuối cùng, thì có 4 cách để chọn màu cho n chiếc đó, và 3 cách để chọn ra một chiếc xe khác màu đặt bên cạnh n chiếc đó. Còn lại $2n - 2 - (n + 1) = n - 3$ vị trí ở giữa, ta có thể chọn một trong 4 màu cho mỗi vị trí. Vậy tổng số cách chọn trong trường hợp này là $2 \times 4 \times 3 \times 4^{n-3}$ (vì có thể đặt n chiếc vào đầu hoặc cuối đường thẳng).

Nếu như n chiếc cùng màu được đặt ở vị trí giữa đường thẳng, thì vẫn có 4 cách chọn màu cho n chiếc này, và có thêm 3 cách chọn màu khác với màu đó cho mỗi chiếc kề trái kề phải của n chiếc đó. Còn $n - 4$ chiếc còn lại, mỗi chiếc có khả năng chọn 4 màu. Vậy tổng số cách chọn cho một đoạn gồm n chiếc cùng màu là $4 \times 3^2 \times 4^{n-4}$. Ngoài ra lại có $n - 3$ vị trí để đặt n chiếc cùng màu vào giữa đoạn, nên tổng số cách chọn trong trường hợp này là: $(n - 3) \times 4 \times 3^2 \times 4^{n-4}$.

Vậy kết quả cuối cùng là: $2 \times 4 \times 3 \times 4^{n-3} + (n - 3) \times 4 \times 3^2 \times 4^{n-4}$. Vì kết quả phải chia dư cho $10^9 + 7$ nên ta sẽ áp dụng giải thuật bình phương và nhân với đồng dư thức.

Cài đặt

Ngôn ngữ C++:

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

using namespace std;

const long long mod = 1e9 + 7;

long long power_modulo(long long a, long long b, long long mod)
{
    if (b == 0)
        return 1LL;

    long long half = power_modulo(a, b / 2, mod) % mod;

    if (b & 1)
        return (((half * half) % mod) * (a % mod)) % mod;
    else
        return (half * half) % mod;
}

main()
{
    int n;
    cin >> n;

    long long x = (24 * power_modulo(4, n - 3, mod)) % mod;
    long long y = (((n - 3) * 36) % mod) * power_modulo(4, n - 4, mod)) % mod;

    cout << (x + y) % mod;
}
```

Ngôn ngữ Python:

```
def power_modulo(a, b, mod):
    if b == 0:
        return 1

    half = power_modulo(a, b // 2, mod) % mod

    if b & 1:
        return (half * half * a) % mod
```

```

else:
    return (half * half) % mod

if __name__ == '__main__':
    n = int(input())

    mod = int(1000000007)
    x = (24 * power_modulo(4, n - 3, mod)) % mod
    y = (((n - 3) * 36) % mod) * power_modulo(4, n - 4, mod)) % mod

    print((x + y) % mod)

```

IV. Tài liệu tham khảo

- <https://cowboycoder.tech/article/phep-nhan-an-do-va-phep-tinh-luy-thua>
- <https://vnoi.info/wiki/translate/he/Number-Theory-3.md>
- <https://cp-algorithms.com/algebra/binary-exp.html>

Buổi 4 .Hệ cơ số

I. Giới thiệu chung

Trong Toán học, hệ cơ số (hay hệ đếm) là một hệ thống các kí hiệu toán học và quy tắc sử dụng tập kí hiệu đó để biểu diễn số đếm. Các kí hiệu toán học có thể là chữ số hoặc các kí tự chữ cái. Cần phân biệt giữa **Hệ cơ số** và **Cơ số** (số lượng kí hiệu sử dụng trong một hệ cơ số).

Có rất nhiều hệ cơ số khác nhau, mỗi hệ cơ số có những quy tắc biểu diễn số khác nhau. Những dãy kí hiệu giống nhau có thể sẽ ứng với những số khác nhau trong các hệ đếm khác nhau. Ví dụ trong hệ thập phân, 11 thể hiện số “mười một”, tuy nhiên trong hệ nhị phân, nó lại thể hiện số “ba”,... Số đếm mà dãy kí hiệu thể hiện được gọi là giá trị của nó.

Có hai loại hệ cơ số là hệ cơ số phụ thuộc vào vị trí và hệ cơ số không phụ thuộc vào vị trí. Chẳng hạn, hệ đếm La Mã là một hệ cơ số không phụ thuộc vào vị trí. Hệ đếm này gồm các kí hiệu chữ cái: *I, V, X, L, C, D, M*; mỗi kí hiệu có giá trị cụ thể:

$$I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000$$

Trong hệ đếm này, giá trị của các kí hiệu không phụ thuộc vào vị trí của nó. Ví dụ, trong hai biểu diễn IX (9) và XI (11) thì X đều có giá trị là 10.

Các hệ đếm thường dùng là các hệ đếm phụ thuộc vị trí. Mọi số nguyên $base$ bất kỳ có giá trị lớn hơn 1 đều có thể được chọn làm cơ số cho một hệ đếm. Trong các hệ đếm loại này, số lượng kí hiệu sử dụng sẽ chính bằng cơ số của hệ đếm đó, và giá trị tương ứng của các kí hiệu là: $0, 1, 2, \dots, base - 1$. Để thể hiện một biểu diễn X là biểu diễn của số ở hệ cơ số $base$, ta kí hiệu là X_{base} .

II. Biểu diễn số trong các hệ đếm

1. Giá trị của một số trong hệ cơ số bất kỳ

Trong một hệ cơ số b , giả sử số N có biểu diễn:

$$d_n d_{n-1} d_{n-2} \dots d_0, d_{-1} d_{-2} \dots d_{-m}$$

Trong đó có $n + 1$ chữ số bên trái dấu phẩy, m chữ số bên phải dấu phẩy thể hiện cho phần nguyên và phần phân của N , và $0 \leq d_i < b$. Khi đó giá trị của số N được tính theo công thức:

$$N = d_n b^n + d_{n-1} b^{n-1} + \dots + d_0 b^0 + d_{-1} b^{-1} + d_{-2} b^{-2} + \dots + d_{-m} b^{-m}$$

Giá trị của một số trong hệ cơ số b cũng chính là biểu diễn tương ứng của nó ở hệ cơ số 10.

Cài đặt

Chương trình tính giá trị một số thực N trong hệ cơ số b . Ta có thể làm cách đơn giản hơn như sau: Coi như số đó không có phần phân, tính giá trị của nó trong hệ cơ số b rồi chia cho 10^x , với x là số chữ số phần phân.

Ngôn ngữ C++:

```
void enter(string &N, int &b)
{
```

```

getline(cin, N); // Nhập N ở dạng xâu.
cin >> b;
}

// Tính giá trị của biểu diễn N trong hệ cơ số'b, chính là biểu diễn thập phân c
double get_value(string &N, int b)
{
    int pos = N.find('.'); // Tìm vị trí đầu '.' của N.
    long long mul = (long long) pow(10, N.size() - pos - 1);

    // Tính giá trị toàn bộ số'N, coi như không có phần phân.
    double value = 0, power = 1;
    for (int i = N.size() - 1; i >= 0; --i)
        if (N[i] != '.')
    {
        value += (double) (N[i] - '0') * power;
        power = power * (double) b;
    }

    // Kết quả là lấy giá trị đã tính chia cho 10^x với x là số'chữ số'phân phâ
    return value / mul;
}

```

```

main()
{
    string N;
    int b;

    enter(N, b);
    solution(N, b);
}

```

Ngôn ngữ Python:

```

def enter(N, b):
    N = input()
    b = int(input())

def get_value(N, b):
    pos = N.find(".")
    mul = b ** (len(s) - pos - 1) if pos >= 0 else 1

    res, power = 0, 1

```

```

for d in N[::-1]:
    if d != ".":
        res += power * int(d)
        power *= b

return res / mul

if __name__ == '__main__':
    N = ""
    b = 0

    enter(N, b)
    get_value(N, b)

```

2. Các hệ cơ số thông dụng trong Tin học

Trong Tin học, ngoài hệ cơ số 10, người ta còn sử dụng hai loại hệ đếm sau:

- Hệ cơ số 2 (Hệ nhị phân): Chỉ sử dụng hai kí hiệu 0 và 1. Lấy ví dụ, $1011_2 = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 11_{10}$.
- Hệ cơ số 16 (Hệ thập lục phân hay hệ Hexa): Sử dụng các chữ số từ 0 tới 9 và 6 chữ cái A, B, C, D, E, F ; trong đó A, B, C, D, E, F có giá trị lần lượt là 10, 11, 12, 13, 14, 15. Lấy ví dụ, $16A = 10 \times 16^0 + 6 \times 16^1 + 1 \times 16^2 = 362_{10}$.

3. Biểu diễn số nguyên và số thực

3.1. Biểu diễn số nguyên

Trong Tin học, các số nguyên có thể được biểu diễn dưới dạng có dấu hoặc không dấu. Để biểu diễn số nguyên, ta có thể chọn kích thước số nguyên là 1 byte, 2 byte, 4 byte, ..., 2^N byte, mỗi cách chọn sẽ tương ứng với một khoảng giá trị có thể biểu diễn.

Đối với số nguyên không âm, kích thước 2^N byte sẽ lưu trữ được các số trong phạm vi từ 0 tới $(2^{8 \times 2^N} - 1)$, bởi 1 byte gồm 8 bit và toàn bộ các bit đều được sử dụng để biểu diễn giá trị số.

Đối với số nguyên có dấu, bit bên phải nhất của số nguyên sẽ được dùng để thể hiện dấu của số đó, quy ước 1 là dấu âm, 0 là dấu dương, các bit còn lại sẽ dùng để biểu diễn giá trị số. Theo đó, số nguyên kích thước 2^N sẽ biểu diễn được các giá trị trong phạm vi $(-2^{8 \times 2^N - 1} + 1)$ đến $(2^{8 \times 2^N - 1} - 1)$. Vấn đề này sẽ liên quan tới việc lựa chọn **kiểu dữ liệu** và kiểm soát bộ nhớ chương trình khi lập trình.

bit $8N - 1$ (bit dấu)	bit $8N - 2$	bit $8N - 3$...	bit 2	bit 1	bit 0
------------------------	--------------	--------------	-----	-------	-------	-------

Biểu diễn số nguyên kích thước N byte

3.2. Biểu diễn số thực

Khác với cách viết trong Toán học, khi mà ta dùng dấu , để ngăn cách giữa phần nguyên và phần phân; trong Tin học ta thay dấu , bằng dấu ., và các nhóm ba chữ số cạnh nhau sẽ viết liền. Ví dụ, trong toán ta viết 123 456, 789; thì trong Tin học sẽ viết là 123456.789.

Một cách biểu diễn mà máy tính sử dụng để lưu trữ số thực là dạng **khoa học**, khi mọi số thực sẽ được biểu diễn ở dạng $\pm M \times 10^{\pm K}$. Trong đó, $0, 1 \leq M < 1$, M được gọi là **phần định trị**, và K là một số nguyên không âm được gọi là **phần bậc**. Ví dụ, số 123 456, 789 sẽ được biểu diễn dưới dạng **khoa học** là 0.123456789×10^6 .

4. Phân tách các chữ số của một số nguyên

Việc đếm số chữ số của một số nguyên dương N không có gì khó khăn, bởi vì các số nguyên đều có thể coi như biểu diễn dưới dạng thập phân. Vì thế, ta sẽ chia N cho 10 tới khi thương bằng 0, số lần chia sẽ tương ứng với số chữ số của N .

Cài đặt 1: Đếm số chữ số của số nguyên dương N theo cách thủ công

Ngôn ngữ C++:

```
int cnt_digits(int n)
{
    // Xét riêng trường hợp n = 0.
    if (n == 0)
        return 1;
```

```

int digits = 0;
while (n != 0)
{
    ++digits;
    n /= 10;
}

return digits;
}

```

Ngôn ngữ Python:

```

def cnt_digits(N):
    digits = 0
    while N != 0:
        digits += 1
        N /= 10

    return digits

```

Tuy nhiên, hãy giả sử số nguyên N có n chữ số được biểu diễn ở hệ thập phân dưới dạng:

$$N = d_n d_{n-1} d_{n-2} \dots d_1$$

Phân tích cấu tạo số của N , ta có:

$$N = d_n \times 10^{n-1} + d_{n-1} \times 10^{n-2} + \dots + d_1 \times 10^0$$

$$\Rightarrow \log(d_n \times 10^{n-1}) \leq \log(N) = \log(d_n \times 10^{n-1} + d_{n-1} \times 10^{n-2} + \dots + d_1 \times 10^0) \leq \log(10^n)$$

$$\Leftrightarrow (n - 1) \leq \log(N) \leq n.$$

$$\Leftrightarrow \log(N) \leq n \leq \log(N) + 1.$$

Giữa hai số $\log(N)$ và $\log(N) + 1$ chỉ có duy nhất một số là $\lfloor \log(N) \rfloor + 1$. Vậy $n = \lfloor \log(N) \rfloor + 1$.

Khi đó, các bạn có thể sử dụng trực tiếp hàm `log10()` của thư viện `<cmath>` trong C++, hàm `log()` trong thư viện `math` của Python để đếm số chữ số của N .

Cài đặt 2: Đếm số chữ số của số nguyên dương N bằng hàm `log`

Ngôn ngữ C++:

```
#include <cmath>

using namespace std;

int cnt_digits(int N)
{
    return (int) log10(N) + 1;
}
```

Ngôn ngữ Python:

```
import math

def cnt_digits(N):
    return int(log(N)) + 1
```

Dựa vào biểu diễn trên của số nguyên N , ta nhận thấy, chữ số hàng đơn vị của N chính bằng $N \bmod 10$, chữ số hàng chục bằng $N \bmod 100, \dots$, chữ số ở hàng thứ K tính từ phải qua trái chính bằng $N \bmod 10^K$. Đối với bất kỳ hệ cơ số nào ta cũng có thể coi như đang ở hệ cơ số 10 để tìm các chữ số từ phải qua trái theo cách này.

Cài đặt 3: Xác định chữ số thứ K từ bên phải sang của số nguyên dương N

Ngôn ngữ C++:

```
// Tìm chữ số thứ K từ bên phải sang của số nguyên dương N.
int find_k_digits(int N, int K)
{
    int power = (int) pow(10, K);

    return N % power;
}
```

Ngôn ngữ Python:

```
# Tìm chữ số thứ K từ bên phải sang của số nguyên dương N.
def find_k_digits(N, K):
    power = 10 ** K
```

```
return N % power
```

III. Chuyển đổi giữa các hệ cơ số

1. Chuyển đổi từ hệ cơ số 10 sang các hệ cơ số khác

Xét số thực N ở hệ cơ số 10. Để tìm biểu diễn của N trong hệ cơ số b , ta sẽ lần lượt chuyển đổi phần nguyên và phần phân, sau đó ghép chúng lại với nhau.

1.1. Chuyển đổi phần nguyên

Xét phần nguyên của N là K . Giả sử trong hệ đếm b , K có giá trị là:

$$K = d_n b^n + d_{n-1} b^{n-1} + \cdots + d_1 b + d_0$$

Do $0 \leq d_0 < b$ nên khi chia K cho b thì phần dư của phép chia là d_0 , còn thương là:

$$K_1 = d_n b^n + d_{n-1} b^{n-1} + \cdots + d_1$$

Tương tự, d_1 chính là phần dư của phép chia K_1 cho b , và sẽ thu được K_2 là thương của phép chia đó. Lặp lại quá trình chia như trên tới khi thu được thương bằng 0, khi đó biểu diễn của K ở hệ cơ số b là $d_n \dots d_0$. Nói cách khác, ta lấy K chia cho b , thu nhận thương và số dư ở mỗi lần chia cho tới khi thương bằng 0, khi đó viết các số dư theo thứ tự ngược từ dưới lên trên thì sẽ thu được biểu diễn của K ở hệ cơ số b .

Ví dụ, với $K = 4_{10}$ và $b = 2$, quá trình chuyển đổi sẽ diễn ra như sau:

- Chia 4 cho 2, thu được $K_1 = 2, d_0 = 0$.
- Chia 2 cho 2, thu được $K_2 = 1, d_1 = 0$.
- Chia 1 cho 2, thu được $K_3 = 0, d_2 = 1$.
- Tới đây quá trình dừng lại, thu được kết quả $4_{10} = 100_2$.

Cài đặt

Ngôn ngữ C++:

```

// Chuyển phần nguyên K của số N sang hệ đếm b, lưu vào chuỗi res.
string change_integer_path(int K, int b)
{
    string res;
    while (K != 0)
    {
        res = (char) (K % b + 48) + res;
        K /= b;
    }

    return res;
}

```

Ngôn ngữ Python:

```

# Chuyển phần nguyên K của số N sang hệ đếm b, lưu vào chuỗi res.
def change_integer_path(K, b):
    res = ""
    while K != 0:
        res = str(K % b) + res
        K /= b

    return res

```

1.2. Chuyển đổi phần phân

Xét phần phân của số thực N là K' . Giả sử trong hệ đếm b , K' có giá trị là:

$$K' = d_{-1}b^{-1} + d_{-2}b^{-2} + \cdots + d_{-m}b^{-m} \quad (1)$$

Nhân cả 2 vế của (1) với b , ta có:

$$K'_1 = d_{-1} + d_{-2}b^{-1} + \cdots + d_{-m}b^{-(m-1)}$$

Ta thấy, d_{-1} chính là phần nguyên của kết quả phép nhân K' với b , còn phần phân của kết quả là:

$$K'_2 = d_{-2}b^{-1} + \cdots + d_{-m}b^{-(m-1)} \quad (2)$$

Tiếp tục lặp lại phép nhân như trên với (2), ta sẽ thu được d_{-2} là phần nguyên. Làm liên tục theo cách này, cuối cùng thu được dãy $d_{-1}d_{-2}d_{-3}\dots$, trong đó $0 \leq d_i < b$. Nói cách khác, lấy phần phân K' nhân liên tục với b , ở mỗi bước thu nhận chữ số ở phần nguyên của kết quả

và lặp lại quá trình nhân với phần phân của kết quả cho tới khi thu được số lượng chữ số như ý muốn.

Ví dụ, với $K' = 0.123_{10}$, $b = 2$ và cần lấy tới 5 chữ số phần phân ở kết quả, ta làm như sau:

- $0.123 \times 2 = 0.246 \rightarrow d_{-1} = 0.$
 - $0.246 \times 2 = 0.492 \rightarrow d_{-2} = 0.$
 - $0.492 \times 2 = 0.984 \rightarrow d_{-3} = 0.$
 - $0.984 \times 2 = 1.968 \rightarrow d_{-4} = 1.$
 - $0.968 \times 2 = 1.936 \rightarrow d_{-5} = 1.$
- ...

Tới đây thu được kết quả $0.123_{10} = 0.00011..._2$

Cài đặt

Ngôn ngữ C++:

```
// Chuyển phân số K sang hệ đếm b, lấy cnt_digits chữ số phân.
string change_double_path(double K, int b, int cnt_digits)
{
    string ans;
    while (ans.size() < cnt_digits)
    {
        double next_K = K * (double) b;
        int digit = (int) next_K;

        ans = ans + (char) (digit + 48);
        K = next_K - (int) next_K;
    }

    return ans;
}
```

Ngôn ngữ Python:

```
# Chuyển phân số K sang hệ đếm b, lấy cnt_digits chữ số phân.
def change_double_path(K, b, cnt_digits):
    res = []
    while len(res) < cnt_digits:
        next_K = K * float(b)
        digit = int(next_K)
```

```

    res.append(digit)
    K = next_K - int(next_K)

    return res

```

2. Chuyển đổi số từ hệ cơ số x sang hệ cơ số y

Để chuyển một số N từ hệ cơ số x sang hệ cơ số y , ta làm theo các bước sau:

- Bước 1: Tính giá trị của số N trong hệ cơ số x , nói cách khác là chuyển N_x sang hệ cơ số 10.
- Bước 2: Chuyển kết quả vừa tìm được sang hệ cơ số y theo phương pháp chuyển một số ở hệ 10 sang hệ y ở phần 1.

Cài đặt

Tái sử dụng lại một số hàm đã xây dựng sẵn ở trên: `get_value()`, `change_integer_path()`, `change_double_path`, ta sẽ chuyển đổi được số thực N từ hệ cơ số x sang hệ cơ số y .

Ngôn ngữ C++:

```

// Chuyển số thực N từ hệ đếm x sang hệ đếm y, lấy d chữ số sau dấu phẩy.
string change_x_to_y(double N, int x, int y, int d)
{
    string NN = to_string(N);
    double value_x = get_value(NN, x);

    int integer_path = (int) value_x;
    double double_path = value_x - integer_path;

    string res = change_integer_path(integer_path, y) + '.'
                 + change_double_path(double_path, y, d);

    return res;
}

```

Ngôn ngữ Python:

```

def change_x_to_y(N, x, y, d):
    NN = str(N)

```

```

value_x = get_value(NN, x)

integer_path = int(value_x)
double_path = value_x - integer_path

res = change_integer_path(integer_path, y) + '.'
    + change_double_path(double_path, y, d)

return res

```

3. Chuyển đổi giữa hệ cơ số 2 (hệ nhị phân) và hệ cơ số 16 (hệ Hexa)

Do 16 là lũy thừa của 2 ($16 = 2^4$), nên việc chuyển đổi giữa hệ nhị phân và hệ hexa có thể được thực hiện dễ dàng theo quy tắc sau:

- Bước 1: Tính từ vị trí phân cách phần nguyên và phần phân, ta gộp các chữ số thành từng nhóm 4 chữ số về hai phía trái phải, nếu thiếu chữ số sẽ thay bằng các chữ số 0.
- Bước 2: Tính giá trị của từng nhóm chữ số, sau đó thay kết quả bằng một kí hiệu tương ứng ở hệ Hexa. Ví dụ 2 tương ứng với 2, 10 tương ứng với A,...
- Bước 3: Đặt các kí hiệu sau khi đã chuyển đổi vào đúng thứ tự của từng nhóm, ta thu được kết quả chuyển đổi.

Cài đặt 1: Chuyển đổi từ hệ nhị phân sang hệ Hexa

```

#include <bits/stdc++.h>

using namespace std;

const char hexa[16] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                      'A', 'B', 'C', 'D', 'E', 'F'};

string binary_to_hexa(double N)
{
    string NN = to_string(N);
    int pos = NN.find('.');
    string left_path = NN.substr(0, pos), right_path = NN.substr(pos + 1, NN.size()

    // Bổ sung đủ chữ số 0 để tạo thành các nhóm 4.
    while (left_path.size() % 4 != 0)

```

```

left_path = '0' + left_path;
while (right_path.size() % 4 != 0)
    right_path = right_path + '0';

string ans_left, ans_right;
for (int i = 0; i < left_path.size() - 3; i += 4)
{
    // Gộp cụm 4 kí tự liên tiếp.
    string group = left_path.substr(i, 4);

    // Tính giá trị cụm 4 kí tự.
    int power = 1, value = 0;
    for (int j = 3; j >= 0; --j)
    {
        value += (group[j] - '0') * power;
        power *= 2;
    }

    // Lấy kí tự hexa mang giá trị tương ứng.
    ans_left = ans_left + hexa[value];
}

for (int i = 0; i < right_path.size() - 3; ++i)
{
    string group = right_path.substr(i, 4);

    int power = 1, value = 0;
    for (int j = 3; j >= 0; --j)
    {
        value += (group[j] - '0') * power;
        power *= 2;
    }

    ans_right = ans_right + hexa[value];
}

return (ans_left + '.' + ans_right);
}

```

Ngôn ngữ Python:

```

hexa = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        'A', 'B', 'C', 'D', 'E', 'F']

def binary_to_hexa(n: float) -> str:

```

```

nn = str(n)
pos = nn.index('.')
left_path, right_path = nn[ :pos], nn[pos+1:]

# Bổ sung đủ chữ số 0 để tạo thành các nhóm 4.
while len(left_path) % 4 != 0:
    left_path = '0' + left_path
while len(right_path) % 4 != 0:
    right_path = right_path + '0'

ans_left, ans_right = '', ''

for i in range(0, len(left_path)-3, 4):
    # Gộp cụm 4 kí tự liên tiếp
    group = left_path[i:i+4]

    # Tính giá trị cụm 4 kí tự.
    power, value = 1, 0
    for j in range(3, -1, -1):
        value += int(group[j]) * power
        power *= 2

    # Lấy kí tự hexa mang giá trị tương ứng.
    ans_left = ans_left + hexa[value]

for i in range(0, len(right_path)-3, 4):
    group = right_path[i:i+4]

    power, value = 1, 0
    for j in range(3, -1, -1):
        value += int(group[j]) * power
        power *= 2

    ans_right = ans_right + hexa[value]

return ans_left + '.' + ans_right

```

Ở chiều hướng ngược lại, khi chuyển từ hệ nhị phân sang hệ hexa, chúng ta chỉ cần đổi từng kí tự hexa sang cụm bốn kí tự nhị phân có giá trị tương ứng. Ta có thể đơn giản hóa bằng cách khởi tạo trước một mảng binary_code gồm 15 phần tử, với nghĩa binary_code[i] là biểu diễn nhị phân tương ứng với kí tự Hexe i ($0 \leq i \leq 15$). Lưu ý rằng, với $i > 10$ thì sẽ tương ứng với các kí tự A, B, C, D, E, F nên ta cần xử lý tinh tế để biết được kí tự chữ cái tương ứng với i bằng bao nhiêu.

Cài đặt 2: Chuyển đổi từ hệ Hexa sang hệ nhị phân

Ngôn ngữ C++:

```
string hexa_to_binary(double N)
{
    string binary_code[15] = {"0000", "0001", "0010", "0011", "0100", "0101", "0110",
                             "0111", "1000", "1001", "1010", "1011", "1100", "1101", "1110",
                             "1111"};
    string NN = to_string(N);

    string res;
    for (int i = 0; i < NN.size(); ++i)
        if ('0' <= NN[i] && NN[i] <= '9')
            res += binary_code[NN[i] - '0'];
        else if ('A' <= NN[i] && NN[i] <= 'F')
        {
            int pos = NN[i] - 55;
            res += binary[pos];
        }

    return res;
}
```

Ngôn ngữ Python:

```
def hexa_to_binary(n: float) -> str:
    binary_code = ["0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
                  "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"]

    nn = str(n)
    res = ''
    for i in range(len(nn)):
        if '0' <= nn[i] and nn[i] <= '9':
            res += binary_code[nn[i] - '0']
        elif 'A' <= nn[i] and nn[i] <= 'F':
            pos = nn[i] - 55
            res += binary_code[pos]

    return res
```

IV. Bài toán minh họa

1. Số nhị phân thứ K

Đề bài

Xét các số nhị phân có độ dài N với số bé nhất là $\overline{100 \dots 0}$ ($N - 1$ chữ số 0) và số lớn nhất là $\overline{111 \dots 1}$ (N chữ số 1). Ví dụ với $N = 3$, ta có các số nhị phân độ dài 3 là 100, 101, 110 và 111.

Yêu cầu: Cho trước hai số nguyên dương N và K . Hãy xác định số nhị phân thứ K trong dãy số nhị phân có độ dài N ?

Input:

- Một dòng duy nhất chứa hai số nguyên dương N và K .

Ràng buộc:

- $1 \leq N \leq 30$.
- $1 \leq K \leq 10^9$.

Output:

- Số nhị phân thứ K tìm được.

Sample Input:

3 3

Sample Output:

110

Ý tưởng

Trong dãy nhị phân có độ dài N , số bé nhất là $\overline{100 \dots 0}$ ($N - 1$ chữ số 0). Số này có giá trị tương ứng trong hệ thập phân chính là 2^{N-1} . Muốn lấy số thứ K , ta chỉ cần cộng thêm

$K - 1$ đơn vị vào số bé nhất đó, nhưng nếu như tiến hành cộng ở hệ nhị phân thì sẽ khá phức tạp. Do đó, ta có thể lấy luôn giá trị $2^{N-1} + (K - 1)$ ở hệ thập phân của số nhị phân thứ K , rồi đổi ngược lại hệ nhị phân, kết quả vẫn sẽ hoàn toàn chính xác.

Độ phức tạp: $O(N)$.

Code mẫu

```
#include <bits/stdc++.h>
#define int long long

using namespace std;

void solution(int n, int k)
{
    int power = 1 << (n - 1) + (k - 1); // Tính  $2^{n - 1} + (k - 1)$ .

    string res;
    while (power > 0)
    {
        if (power % 2)
            res = '1' + res;
        else
            res = '0' + res;

        power /= 2;
    }

    cout << res;
}

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

    int n, k;
    cin >> n >> k;

    solution(n, k);

    return 0;
}
```

2. Biểu diễn nhị phân

Đề bài

Mọi số nguyên dương X đều có thể biểu diễn trong hệ nhị phân tương tự như biểu diễn trong hệ thập phân. Chẳng hạn, số $X = 17$ có biểu diễn nhị phân là 10001 vì $17 = 1 \times 2^4 + 1$.

Yêu cầu: Cho trước một số nguyên dương X . Hãy thực hiện các yêu cầu sau:

- Tìm biểu diễn nhị phân của số X .
- Tìm số Y lớn nhất trong hệ thập phân sao cho biểu diễn nhị phân của Y thu được từ X bằng cách hoán vị vòng quanh các chữ số trong biểu diễn nhị phân của X .

Input:

- Gồm một số nguyên dương X duy nhất.

Ràng buộc:

- $1 \leq X \leq 10^9$.

Output:

- Dòng thứ nhất ghi biểu diễn nhị phân của X .
- Dòng thứ hai ghi số Y tìm được.

Sample Input:

17

Sample Output:

10001
24

Giải thích:

Ta có $17 = 1 \times 2^4 + 1$, do đó biểu diễn nhị phân của 17 là 10001 .

Số nhị phân có giá trị lớn nhất thu được từ X khi hoán vị vòng quanh các chữ số trong biểu diễn nhị phân của X là 11000. Số đó có giá trị thập phân là 24.

Ý tưởng

Đối với yêu cầu thứ nhất, ta dùng thuật toán chuyển đổi từ hệ cơ số 10 sang hệ nhị phân thông thường, không có gì đặc biệt. Sau đó lưu kết quả thu được vào một xâu s .

Đối với yêu cầu thứ hai, trước tiên các bạn cần hiểu rõ thế nào là **hoán vị vòng quanh**? Bởi vì có rất nhiều bạn ở bài này sẽ lầm tưởng kết quả là đảo toàn bộ số 1 lên trước, số 0 về cuối. Tuy nhiên, ta chỉ được phép xét các **hoán vị vòng quanh** của xâu nhị phân s , tức là cứ đảo một chữ số từ trên đầu xuống cuối thì ta được một hoán vị vòng quanh, chứ không phải hoán vị lộn xộn tất cả các chữ số. Như vậy, nếu xâu nhị phân có độ dài n thì ta sẽ có n hoán vị vòng quanh.

Để xét các hoán vị vòng quanh, ta sử dụng một kĩ thuật nhỏ, đó là nhân đôi xâu. Chẳng hạn, xâu 110011 sẽ trở thành 11001110011. Gọi n là độ dài của xâu nhị phân cũ, ta xét từng vị trí i ($0 \leq i < n$), thì một xâu con độ dài n bắt đầu từ vị trí i sẽ là một hoán vị vòng quanh. Sau đó, với mỗi hoán vị này ta chỉ cần đổi nó sang hệ thập phân lại, rồi lấy kết quả lớn nhất là xong.

Độ phức tạp: $O(n^2)$ với n là độ dài xâu nhị phân s .

Code mẫu

```
#include <bits/stdc++.h>
#define int long long

using namespace std;

string dec_to_bin(int x)
{
    string res;

    while (x != 0)
    {
        res = (char) (x % 2 + '0') + res;
        x /= 2;
    }

    return res;
```

```

}

// Tìm giá trị thập phân của một xâu nhị phân S.
int bin_to_dec(string s)
{
    int exp = 1, res = 0;

    for (int i = s.size() - 1; i >= 0; --i)
    {
        res = res + (s[i] - '0') * exp;
        exp *= 2;
    }

    return res;
}

/***
 * Hàm tính toán hai yêu cầu của đề bài.
 * Yêu cầu thứ nhất: Đưa ra biểu diễn nhị phân của số X -> Dùng hàm dec_to_bin()
 * Yêu cầu thứ hai: Ta xét mọi hoán vị vòng quanh của xâu nhị phân s (là biểu c
     thập phân lớn nhất trong tất cả các hoán vị vòng quanh đó. Phương pháp xây c
     đôi xâu lên, rồi xét mọi xâu độ dài n (n là độ dài xâu np ban đầu) bắt đầu t
*/
void solution(int x)
{
    string circular_per = dec_to_bin(x);

    // Kết thúc yêu cầu thứ nhất: In ra biểu diễn nhị phân của x.
    cout << circular_per << endl;

    // Bắt đầu yêu cầu thứ hai, ta nhân đôi xâu nhị phân lên để tiến hành tìm c
    int n = circular_per.size(), max_decimal = 0;
    circular_per += circular_per;

    for (int i = 0; i < n; ++i)
        max_decimal = max(max_decimal, bin_to_dec(circular_per.substr(i, n)));

    cout << max_decimal;
}

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

    int x;

```

```
cin >> x;  
  
solution(x);  
  
return 0;  
}
```

V. Tài liệu tham khảo

- https://en.wikipedia.org/wiki/Numeral_system
- [Sách giáo khoa Tin học 10 - Thầy Hồ Sĩ Đàm](#)

Buổi 5. Bài toán tìm kiếm

1. Tìm kiếm - một khái niệm quen thuộc trong cuộc sống

Có bao giờ bạn phải đau đầu vì để quên chiếc ví ở đâu đó trong nhà mà tìm mãi không thấy? Hay việc các bạn nữ luôn không thể nào tìm thấy bộ quần áo phù hợp để lên phố mặc dù số lượng trang phục xếp nặng trĩu trong tủ quần áo?

Cuộc sống chúng ta luôn gắn liền với việc tìm kiếm. Từ một đứa trẻ tò mò tìm kiếm khám phá từng điều thú vị của thế giới đến khi trưởng thành chúng ta phải chật vật tìm kiếm một nửa mảnh ghép còn lại của đời mình, mà ... có nhiều người còn phải chịu thua số phận và đành cam lòng gắn lên mình danh hiệu “FA”.

Đừng nản chí, để giúp bạn giải quyết những vấn đề đó, hôm nay chúng ta sẽ cùng khám phá về thuật toán tìm kiếm trong lập trình - một giải thuật quen thuộc, hữu ích vô cùng trong Tin học mà còn có thể áp dụng ra đời sống, giúp bạn tìm thấy chân ái của đời mình!

2. Bài toán tìm kiếm trong Tin học

Cùng so sánh hai sự việc sau đây:

- Bạn đang cần tìm cục tẩy trong hộp bút.
- Tìm kiếm số 6 ở vị trí nào trong một dãy số cho trước.

Mục tiêu tìm kiếm trong cả hai bài toán đều đã được xác định, đó là “cục tẩy” và “số 6” (cần tìm thấy số 6 xong mới có được vị trí). Và tập “dữ liệu” của chúng ta có (hay phạm vi tìm kiếm) chính là “những đồ vật trong hộp bút” hoặc “các số trong dãy số đã cho trước”. Có thể hiểu, tìm kiếm là quá trình tìm một phần tử nằm trong một tập hợp rất nhiều phần tử dựa vào một yêu cầu nào đó.

Trong Tin học, với sự giúp đỡ của máy tính, rất nhiều thuật toán tìm kiếm đã ra đời với tính hiệu quả ngày càng tăng cao. Những thuật toán tìm kiếm cơ bản nhất có thể kể đến là **Tìm kiếm tuần tự** và **Tìm kiếm nhị phân**. Ngoài ra, áp dụng thêm những cấu trúc dữ liệu trong khi tìm kiếm có thể cho ra những thuật toán có hiệu quả cao hơn nữa.

Bài toán tìm kiếm trong Tin học có thể phát biểu như sau:

"Cho một dãy gồm n đối tượng a_1, a_2, \dots, a_n . Mỗi đối tượng a_i có một khóa key ($1 \leq i \leq n$) gọi là **khóa tìm kiếm**. Cần tìm kiếm đối tượng có khóa bằng k cho trước, tức là $a_i.key = k$ ".

Quá trình tìm kiếm sẽ hoàn thành nếu như có một trong hai trường hợp sau đây xảy ra:

- Tìm được đối tượng có khóa tương ứng bằng k , khi đó phép tìm kiếm thành công.
- Không tìm được đối tượng nào có khóa tương ứng bằng k , khi đó phép tìm kiếm thất bại.

Dưới đây, tôi sẽ trình bày một số thuật toán thông dụng để giải quyết bài toán trên!

II. Giải thuật tìm kiếm tuần tự (Sequential Search)

Ý tưởng

Tìm kiếm tuần tự (Sequential Search hay Linear Search) là một giải thuật đơn giản, rất dễ cài đặt. Bắt đầu từ đối tượng a_1 , duyệt qua tất cả các đối tượng, cho tới khi tìm thấy đối tượng có khóa mong muốn, hoặc duyệt hết toàn bộ dãy mà không tìm thấy khóa đó.

Linear Search

Looking for 4

Data	3	5	1	2	16	90	4	0
Index	0	1	2	3	4	5	6	7

Step : 0

Mô phỏng giải thuật C++:

```
sequential_search(a[], n, k)
{
    for (i = 1; i <= n; ++i)
        if (a[i].key == k)
            return i;

    // Không tìm thấy đối tượng nào có khóa bằng k, trả về -1.
    return -1;
}
```

Đánh giá

Mặc dù giải thuật Tìm kiếm tuần tự rất đơn giản và dễ cài đặt, tuy nhiên nhược điểm của nó nằm ở độ phức tạp. Trong trường hợp tốt nhất, giải thuật có độ phức tạp là $O(1)$, nhưng trong trường hợp xấu nhất lên tới $O(n)$. Vì vậy độ phức tạp tổng quát của giải thuật là $O(n)$, chỉ phù hợp với những bài toán có kích thước không gian tìm kiếm nhỏ.

Ví dụ

Cho một dãy số a gồm n số nguyên a_1, a_2, \dots, a_n ($1 \leq n \leq 1000$). Hãy xác định xem số fibonacci thứ k ($1 \leq k \leq 100$) có xuất hiện trong dãy số hay không, nếu có thì đưa ra vị trí xuất hiện đầu tiên, ngược lại đưa ra -1 .

Cách giải quyết

Trước tiên, ta dùng vòng lặp để tìm ra số fibonacci thứ k , rồi tìm kiếm tuần tự trên dãy số ban đầu để tìm ra vị trí của số fibonacci thứ k trong dãy.

Cài đặt

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

using namespace std;

// Tìm kiếm tuần tự.
int sequential_search(int a[], int n, long long kth_fibo)
{
    for (int i = 1; i <= n; ++i)
        if (a[i] == kth_fibo)
            return i;

    return -1;
}

// Tìm số fibonacci thứ k.
long long find_kth_fibo(int k)
{
    if (k <= 1)
        return k;

    long long f0 = 0, f1 = 1, fk;
    for (int i = 2; i <= k; ++i)
    {
        fk = f0 + f1;
        f0 = f1;
        f1 = fk;
    }

    return fk;
}
```

```

int main()
{
    // Nhập dữ liệu n, k và dãy số.
    int n, k;
    cin >> n >> k;

    int a[n + 1];
    for (int i = 1; i <= n; ++i)
        cin >> a[i];

    cout << sequential_search(a, n, find_kth_fibo(k));
}

```

III. Giải thuật tìm kiếm nhị phân

Ý tưởng

Trước tiên, không gian tìm kiếm cần được sắp xếp lại theo chiều tăng dần hoặc giảm dần của khóa tìm kiếm (mục tiêu là để tạo ra dãy có tính thứ tự). Giả sử dãy đã được sắp xếp tăng dần theo khóa, giải thuật tìm kiếm nhị phân được thực hiện như sau:

- Giả sử cần tìm kiếm trong đoạn $a[l \dots r]$ với khóa tìm kiếm là k , trước hết ta xét phần tử nằm ở giữa dãy là a_{mid} với $mid = \lfloor \frac{l+r}{2} \rfloor$.
- Nếu $a_{mid}.key < k$ thì nghĩa là đoạn từ a_l tới a_{mid} chỉ chứa toàn các đối tượng có khóa nhỏ hơn k , nên ta tiếp tục quá trình tìm kiếm trên đoạn từ a_{mid+1} tới a_r .
- Nếu $a_{mid}.key > k$ thì nghĩa là đoạn từ a_{mid} tới a_r chỉ chứa toàn các đối tượng có khóa lớn hơn k , nên ta tiếp tục quá trình tìm kiếm trên đoạn từ a_l tới a_{mid-1} .
- Nếu $a_{mid}.key = k$ thì quá trình tìm kiếm thành công.

Quá trình tìm kiếm sẽ thất bại nếu như đến một bước nào đó, tập tìm kiếm bị rỗng ($l > r$).

Mô phỏng giải thuật C++:

```

binary_search(a, k)
{
    // Sắp xếp lại các đối tượng tăng dần theo khóa tìm kiếm.
    sort(a);
}

```

```

l = 1, r = n;
// Trong khi tập tìm kiém chưa rỗng th
while (l <= r)
{
    mid = (l + r) / 2;

    // Đã tìm thấy đôi tượng có khóa bằng k, trả về`vị trí.
    if (a[mid].key == k)
        return mid;

    /// Điều chỉnh khoảng tìm kiém cho phù hợp.
    if (a[mid].key < k)
        l = mid + 1;
    else
        r = mid - 1;
}

// Không tìm thấy khóa, trả về`-1.
return -1;
}

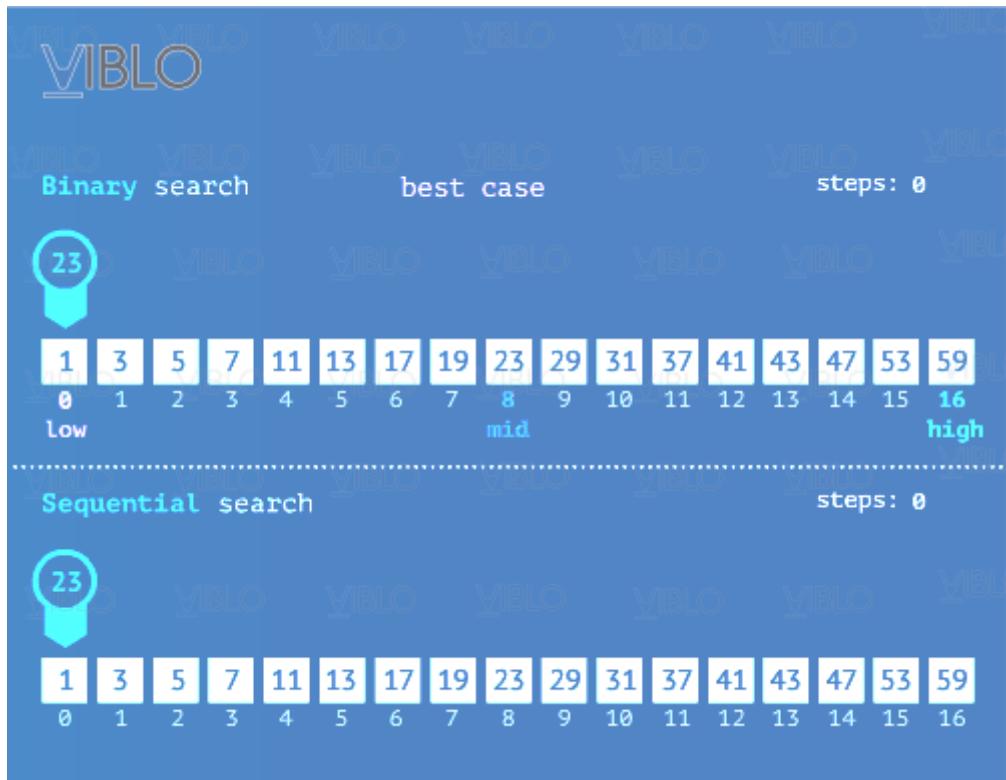
```

Đánh giá

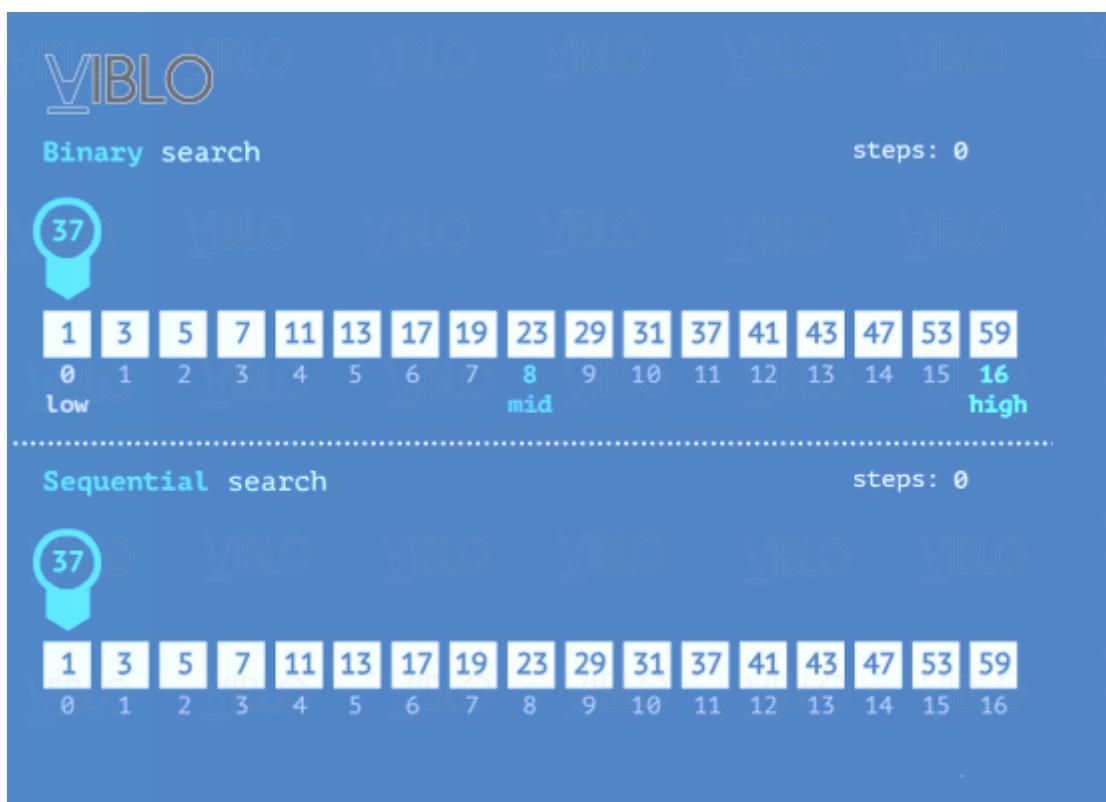
Trong trường hợp tốt nhất, giải thuật Tìm kiếm nhị phân cho ta độ phức tạp $O(1)$. Còn trong trường hợp xấu nhất, do tập tìm kiếm luôn luôn được chia đôi ra, nên số thao tác chỉ mất $O(\log_2(n))$. Vì thế, độ phức tạp tổng quát của giải thuật là $O(\log(n))$.

Tuy nhiên, giải thuật Tìm kiếm nhị phân chỉ có thể thực hiện trên một tập đã sắp xếp, chính vì thế chi phí sắp xếp cũng cần được tính đến. Nếu như dãy số bị thay đổi bởi các thao tác thêm, xóa hay sửa phần tử, thì việc sắp xếp cũng phải thực hiện lại liên tục, từ đó dẫn đến thời gian thực thi bị tăng lên.

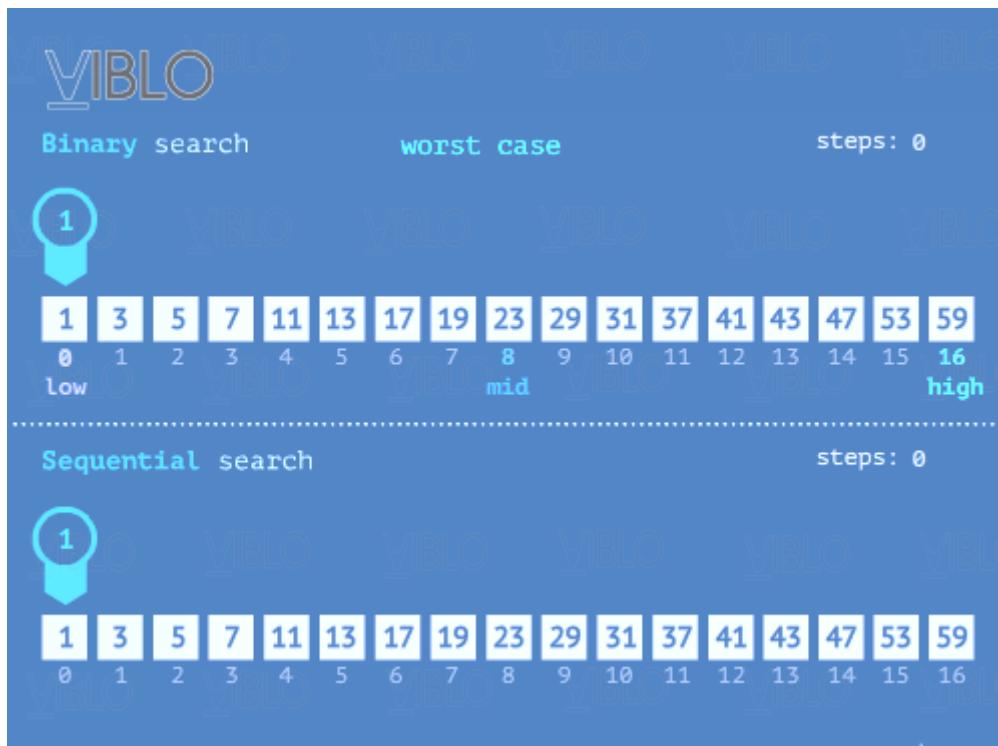
Hình minh họa dưới đây sẽ thể hiện so sánh tương quan giữa hai giải thuật **Sequential Search** và **Binary Search** trong cả ba trường hợp: Tốt nhất, trung bình và xấu nhất:



Trường hợp tốt nhất



Trường hợp trung bình



Trường hợp xấu nhất

Ví dụ

Cho dãy số A gồm n ($1 \leq n \leq 10^5$) phần tử nguyên dương a_1, a_2, \dots, a_n ($a_i \leq 10^9$). Hãy xác định số chính phương nhỏ nhất không xuất hiện trong dãy số?

Cách giải quyết

Với bài toán này, phương pháp đếm phân phối cũng có thể được áp dụng, tuy nhiên người viết sẽ trình bày phương pháp tìm kiếm nhị phân để minh họa cách áp dụng giải thuật.

Đầu tiên, sắp xếp dãy số đã cho theo thứ tự tăng dần. Ta nhận thấy, do $a_i \leq 10^9$ nên $\sqrt{a_i} \leq \sqrt{10^9}$. Vậy chỉ cần duyệt qua các giá trị i từ 0 tới $\sqrt{\max(a_i)} + 1$, sau đó tìm kiếm nhị phân trên dãy xem có tồn tại giá trị i^2 hay không, nếu không thì đó chính là số chính phương nhỏ nhất không xuất hiện trong dãy.

Cài đặt

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

```
using namespace std;
```

```

int a[100001];

int binary_search(int a[], int n, int value)
{
    // Tìm kiêm nhị phân, nếu tìm thấy giá trị value trong dãy số thì trả về true
    int l = 1, r = n;
    while (l <= r)
    {
        int mid = (l + r) / 2;

        if (a[mid] == value)
            return true;

        if (a[mid] < value)
            l = mid + 1;
        else
            r = mid - 1;
    }

    // Nếu không tìm thấy, trả về false.
    return false;
}

int main()
{
    int n;
    cin >> n;

    for (int i = 1; i <= n; ++i)
        cin >> a[i];

    // Sắp xếp mảng tăng dần.
    sort(a + 1, a + n + 1);

    // Tìm giá trị lớn nhất trong mảng.
    int max_value = *max_element(a + 1, a + n + 1);
    // Duyệt các căn bậc hai có thê' tìm kiêm nhị phân bình phương của nó.
    for (int i = 0; i <= sqrt(max_value) + 1; ++i)
        if (!binary_search(a, n, i * i))
        {
            cout << i * i;
            break;
        }
}

```

IV. Tài liệu tham khảo

- <https://www.geeksforgeeks.org/binary-search/>
- [Sách Giải thuật và Lập trình - thầy Lê Minh Hoàng](#)
- https://vi.wikipedia.org/wiki/Tìm_kiếm_nhi_phân
- <https://vnoi.info/wiki/algo/basic/binary-search.md>

Buổi 6, 7 Luyện tập và chữa đẽ

Buổi 8, 9 Quy hoạch động

I. Mở đầu

Trong những bài viết trước, các bạn đã được giới thiệu tuần tự những chiến lược giải thuật từ đơn giản tới nâng cao, như đệ quy, quay lui, nhánh cận, tham lam,...Những chiến lược nói trên thực ra sẽ không xuất hiện quá nhiều trong những cuộc thi lập trình, và cũng không phải là cách hay để giải quyết bài toán, vì nhiều lí do:

- Trong các cuộc thi lập trình, mỗi bài tập đều bị giới hạn thời gian rất chặt. Giải pháp quay lui hay nhánh cận mặc dù luôn luôn cho ra kết quả đúng, nhưng thời gian thực thi quá lớn, nên chỉ có thể lấy được số điểm rất ít.
- Giải pháp Tham lam mặc dù thời gian thực thi khá nhanh, nhưng kết quả lại không phải luôn luôn chính xác.

Còn một giải pháp nữa mà tôi cũng đã giới thiệu tới các bạn, đó là Chia để trị (Divide and Conquer). Tư tưởng của phương pháp này là chia nhỏ bài toán thành các bài toán con đơn giản hơn, tìm lời giải của chúng, cuối cùng kết hợp nghiệm của các bài toán con lại thành nghiệm của bài toán ban đầu. Giải thuật đệ quy cũng được thiết kế dựa trên nguyên lí này. Tuy nhiên, trong quá trình chia nhỏ bài toán lớn, sẽ có rất nhiều bài toán con bị lặp lại, gây ra các bước tính toán thừa thãi. Phương pháp Quy hoạch động ra đời chính là để giải quyết việc đó.

Mặc dù nghe có vẻ đơn giản, nhưng thực tế Quy hoạch động lại là phương pháp tốn rất nhiều giấy mực của các chuyên gia Tin học, cũng như khiến cho những người học lập trình cảm thấy rất đau đầu. Nhưng tỉ lệ thuận với sự phức tạp của nó, thì khả năng ứng dụng của phương pháp này trong các bài toán cũng vô cùng to lớn. Cụ thể ra sao, chúng ta hãy cùng tìm hiểu thông qua bài viết này.

II. Các khái niệm căn bản

1. Công thức truy hồi

Đây là một khái niệm khá quen thuộc với các bạn học sinh giỏi Toán. Đây là một chủ đề quan trọng của **Lý thuyết tổ hợp**, có nhiều ứng dụng trong các bài toán đếm, đồng thời là một công cụ vô cùng hữu hiệu để giải các bài toán có bản chất **Đệ quy**.

Lấy một ví dụ đơn giản, dãy số Fibonacci được định nghĩa đệ quy như sau:

$$\begin{cases} f_0 = f_1 = 1. \\ f_i = f_{i-1} + f_{i-2}; \forall i > 1. \end{cases}$$

Trong định nghĩa trên, ta có hai yếu tố cần chú ý:

- $f_0 = f_1 = 1$. Đây là các giá trị cung cấp sẵn của bài toán, hay còn gọi là **bài toán cơ sở**.
- $f_i = f_{i-1} + f_{i-2}$. Đây là công thức để liên hệ giữa phần tử thứ i của dãy số với các phần tử trước nó, hay còn gọi là **công thức truy hồi**.

Tóm lại, công thức truy hồi là một công thức dùng để liên hệ giữa kết quả của các bài toán con với kết quả của bài toán lớn hơn nó, từ đó tìm ra được lời giải cho bài toán ban đầu. Đây là đặc trưng chỉ xuất hiện trong những bài toán có tính chất đệ quy, nghĩa là lời giải cho bài toán lớn hơn được định nghĩa thông qua lời giải cho bài toán nhỏ hơn nó.

2. Bài toán tối ưu

Bài toán tối ưu là một bài toán có nhiều ứng dụng trong thực tế. Các bạn có lẽ đã nghe nhiều tới khái niệm này trong các phương pháp trước đó, nhưng chỉ là một cách đại thể. Bài toán tối ưu được định nghĩa chính xác như sau:

Xét bài toán có nghiệm là X . Gọi $f(X)$ là một hàm đánh giá độ “tốt” của nghiệm X ; còn (g_1, g_2, \dots, g_n) là các hàm điều kiện của bài toán. Nếu như bài toán yêu cầu tìm một nghiệm X sao cho X thỏa mãn tất cả các hàm điều kiện ($g_i(X) = \text{true}, \forall i : 1 \leq i \leq n$), đồng thời $f(X)$ là “tốt nhất”, thì bài toán đó gọi là bài toán tối ưu.

III. Phương pháp Quy hoạch động

1. Bài toán con gối nhau và Cấu trúc con tối ưu

Bài toán con gối nhau

Như đã nói, những bài toán có bản chất đệ quy đều có thể giải được bằng phương pháp đệ quy. Hãy lấy luôn ví dụ là bài toán dãy Fibonacci ở trên. Nếu như sử dụng phương pháp đệ quy, ta sẽ cài đặt nó như sau:

Ngôn ngữ C++:

```
int fibo(int n)
{
    if (n <= 1)
        return 1;

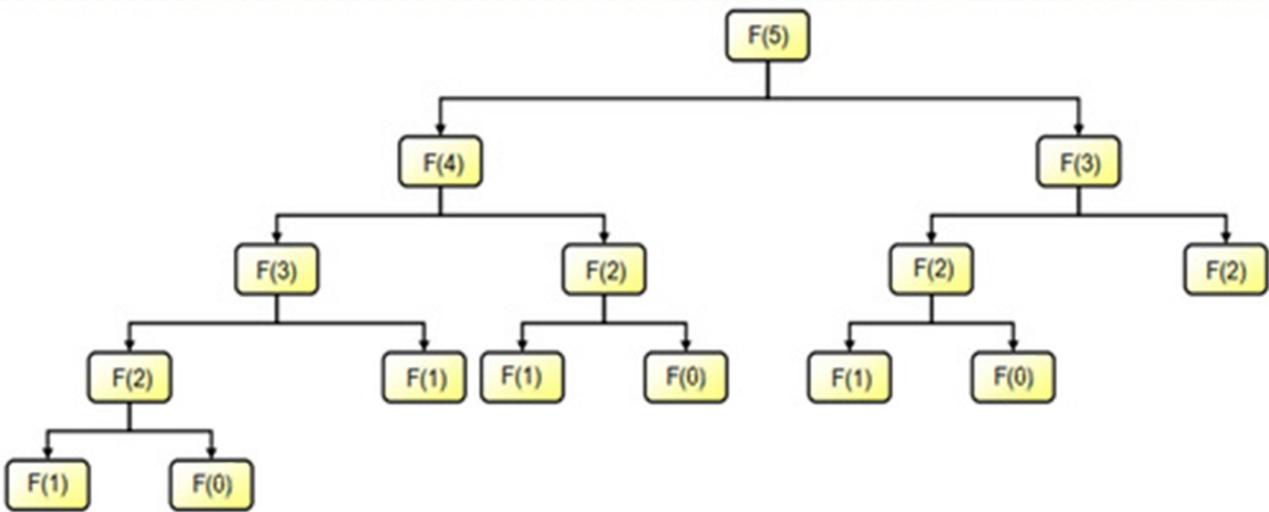
    return fibo(n - 1) + fibo(n - 2);
}
```

Ngôn ngữ Python:

```
def fibo(n):
    if n <= 1:
        return 1

    return fibo(n - 1) + fibo(n - 2)
```

Hãy cùng nhìn vào sơ đồ thực thi của lời giải đệ quy trên, với $n = 5$:



Ta thấy để tính được bài toán $fibo(5)$, chương trình phải tính lại 1 lần $fibo(4)$, 2 lần $fibo(3)$, 3 lần $fibo(2)$, 5 lần $fibo(1)$ và 3 lần $fibo(0)$. Những bài toán trùng lặp đó gọi là **bài toán con gối nhau**, chúng chính là tác nhân chính khiến cho giải thuật đệ quy chạy rất chậm, vì việc tính toán phải thực hiện lại ngay cả khi bài toán đã có đáp số rồi!

Cấu trúc con tối ưu

Một bài toán tối ưu được gọi là thỏa mãn tính chất có “cấu trúc con tối ưu” khi và chỉ khi, lời giải tối ưu cho bài toán đó có thể thu được bằng cách sử dụng lời giải tối ưu của các bài toán con của nó.

Lấy một ví dụ đơn giản, bạn cần tìm một đường đi ngắn nhất từ nhà tới ga Hà Nội, và bạn biết rằng để tới được ga Hà Nội, thì chắc chắn phải đi qua phố Lê Duẩn. Vậy bạn có thể tìm một đường đi ngắn nhất từ nhà tới phố Lê Duẩn, rồi từ phố Lê Duẩn đi tới ga Hà Nội. Việc chia nhỏ các bài toán sẽ lại tiếp tục từ phố Lê Duẩn, cho tới khi tới được một địa điểm nào đó mà bạn đã biết chắc chắn đường đi ngắn nhất từ nhà mình tới địa điểm đó. Có thể nói, trong cuộc sống thường ngày chúng ta cũng hay tư duy theo cấu trúc con tối ưu, chứ không chỉ trong toán học!

2. Phương pháp Quy hoạch động

Quy hoạch động là phương pháp ra đời vào năm 1953, được sáng tạo bởi nhà Toán học người Mỹ Richard Bellman (1920 - 1984) để làm giảm thời gian chạy của các bài toán có tính chất của những **bài toán con gối nhau** và **cấu trúc con tối ưu**.



Richard Bellman

Hai dạng thường gặp nhất của các bài toán giải bằng quy hoạch động là:

- **Bài toán đếm có bản chất đệ quy.**
- **Bài toán tối ưu có bản chất đệ quy.**

Tư tưởng chủ đạo của phương pháp này vẫn là “chia để trị”, tuy nhiên, sự khác biệt là ở phương pháp Quy hoạch động, những bài toán con gối nhau xuất hiện trong quá trình phân rã bài toán lớn sẽ được lưu lại hết lời giải trong một **bảng phương án**, thay vì tính toán lại nhiều lần. Sau đó, các lời giải này sẽ được kết hợp lại với nhau bằng **công thức truy hồi** để tạo thành kết quả của bài toán lớn.

Lấy ví dụ, bài toán Fibonacci thay vì giải bằng phương pháp đệ quy, thì ta có thể lưu lại kết quả của từng bài toán $fibo(i)$ bằng một mảng $f[i]$, rồi sử dụng kết quả đã tính để tạo ra số fibonacci tiếp theo.

Cài đặt C++:

```
int fibo(int n)
{
    int f[n + 1];
    f[0] = f[1] = 1;
```

```

    for (int i = 2; i <= n; ++i)
        f[i] = f[i - 1] + f[i - 2];

    return f[n];
}

```

Cài đặt Python:

```

def fibo(n):
    f = [0] * (n + 1)

    f[0] = f[1] = 1
    for i in range(2, n + 1):
        f[i] = f[i - 1] + f[i - 2]

    return f[n]

```

Với phương pháp này, mỗi giá trị Fibonacci được đảm bảo chỉ phải tính một lần duy nhất, khác hẳn với phương pháp đệ quy đã trình bày ở bên trên. Nhờ thế, thuật toán sẽ thực thi rất hiệu quả về mặt thời gian.

Bảng dưới đây sẽ so sánh những điểm khác biệt giữa hai phương pháp Đệ quy và Quy hoạch động:

Phép phân giải đệ quy	Phương pháp Quy hoạch động
<ul style="list-style-type: none"> - Phân rã bài toán lớn thành nhiều bài toán con, sau đó giải quyết từng bài toán con. - Mỗi bài toán con lại đưa về phân rã thành nhiều bài toán nhỏ hơn và đi tìm lời giải tiếp, bắt kể bài toán đó đã được giải hay chưa. 	<ul style="list-style-type: none"> - Giải các bài toán nhỏ nhất (Bài toán cơ sở), lưu trữ lại kết quả trên một bảng phương án. - Kết hợp kết quả của các bài toán con đã giải để giải quyết các bài toán lớn hơn, rồi tiếp tục lưu lại kết quả. Thực hiện tới khi giải được bài toán lớn nhất. (Sử dụng công thức truy hồi)

Tựu chung lại, một bài toán Quy hoạch động sẽ được giải qua ba bước:

- Bước 1: Giải tất cả các bài toán cơ sở, lưu sẵn vào bảng phương án.
- Bước 2: Sử dụng công thức truy hồi, phối hợp lời giải của các bài toán con, từ đó tìm ra lời giải của bài toán lớn và tiếp tục lưu vào bảng phương án. Thực hiện như vậy tới khi tìm ra lời giải của bài toán ban đầu.

- **Bước 3:** Dựa vào bảng phương án, truy vết tìm ra nghiệm tối ưu của bài toán.

Ngoài ra, các bạn cũng cần ghi nhớ một số khái niệm khi học về phương pháp Quy hoạch động:

- Bài toán giải theo phương pháp Quy hoạch động gọi là **Bài toán Quy hoạch động**.
- Công thức phối hợp nghiệm của các bài toán con để có nghiệm của bài toán lớn gọi là **Công thức truy hồi của Quy hoạch động**.
- Tập các bài toán nhỏ nhất có ngay lời giải để từ đó tìm cách giải quyết các bài toán lớn hơn gọi là **Cơ sở Quy hoạch động**.
- Không gian lưu trữ lời giải các bài toán con để tìm cách phối hợp chúng gọi là **Bảng phương án của Quy hoạch động**.

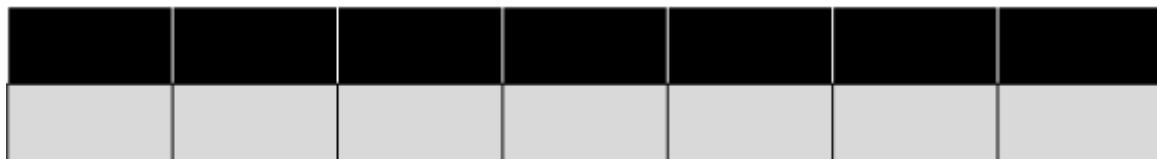
Bây giờ, hãy cùng xét một số bài toán minh họa để hiểu rõ hơn về phương pháp này!

IV. Một số bài toán minh họa

1. Lát gạch

Đề bài

Một hành lang có kích thước $2 \times n$ ô gạch gần được lát gạch hoa. Chẳng hạn với $n = 7$, hành lang được biểu diễn như hình vẽ dưới đây:



Yêu cầu: Hãy đếm số cách lát các viên gạch có kích thước 2×1 sao cho lát kín được hành lang?

Input:

- Một dòng duy nhất chứa số nguyên dương n - độ dài hành lang.

Ràng buộc:

- $1 \leq n \leq 10^6$.

Output:

- Số nguyên duy nhất là số cách lát gạch thỏa mãn. In ra phần dư của kết quả sau khi chia cho $10^9 + 7$.

Sample Input:

4

Sample Output:

5

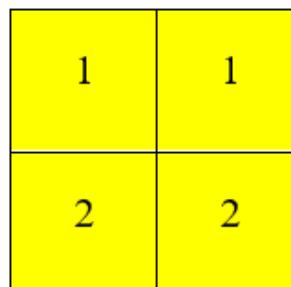
Phân tích ý tưởng

Đầu tiên, ta nhận thấy hai trường hợp $n = 1$ và $n = 2$ có thể giải ngay lập tức:

- Nếu $n = 1$, chỉ có một cách duy nhất để lát gạch là sử dụng một viên gạch đặt theo chiều dọc.
- Nếu $n = 2$, có hai cách để lát gạch là đặt hai viên gạch theo chiều dọc hoặc chiều ngang.

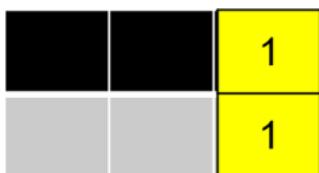


$$n = 1$$

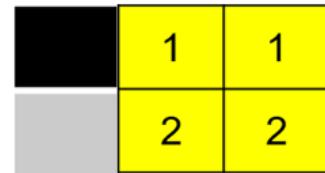


$$n = 2$$

Đặt $dp[i]$ là số cách lát gạch tính đến cột thứ i , thì ta đã biết trước $dp[1] = 1$ và $dp[2] = 2$. Bây giờ, để lát kín cột thứ i của hàng lang, ta có hai phương án như hình bên dưới:



Cách 1: Sử dụng 1 viên gạch 2×1 đặt dọc trên cột thứ i của hành lang.



Cách 2: Sử dụng 2 viên gạch đặt ngang trên hai cột i và $i - 1$ của hành lang

Đối với cách lát gạch thứ nhất, thì tổng số cách lát tính tới cột thứ i sẽ bằng tổng số cách lát tính tới cột thứ $i - 1$. Còn đối với cách lát gạch thứ hai, thì tổng số cách lát tính tới cột thứ i sẽ bằng tổng số cách lát tính tới cột thứ $i - 2$.

Từ đây, ta rút ra công thức truy hồi:

$$dp[i] = dp[i - 1] + dp[i - 2]; \forall i \geq 3$$

Đây là một bài toán đếm có bản chất đệ quy. Nếu sử dụng phương pháp đệ quy, thì giống như bài toán dãy Fibonacci, sẽ có rất nhiều giá trị $dp[i]$ bị tính lại, dẫn đến thời gian thực hiện không thể trong 1 giây được. Giải pháp là sử dụng một mảng $dp[]$ để lưu lại kết quả sau mỗi lần tính, để đảm bảo mỗi giá trị $dp[i]$ chỉ phải tính một lần duy nhất.

Lưu ý để bài yêu cầu in ra kết quả sau khi chia lấy dư cho $10^9 + 7$. Đối với C++, cần thực hiện phép chia dư liên tục để tránh bị tràn số.

Cài đặt

Cài đặt C++:

```
#include <bits/stdc++.h>
#define int long long

using namespace std;

const long long mod = 1e9 + 7;

main()
{
    int n;
    cin >> n;

    vector < int > dp(n + 1);
    dp[1] = 1;
```

```

dp[2] = 2;

for (int i = 3; i <= n; ++i)
    dp[i] = (dp[i - 1] + dp[i - 2]) % mod;

cout << dp[n];

return 0;
}

```

Cài đặt Python:

```

if __name__ == '__main__':
    n = int(input())

    dp = [0] * (n + 1)
    dp[1] = 1
    dp[2] = 2

    for i in range(3, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    mod = 10 ** 9
    print(dp[n] % mod)

```

Đánh giá độ phức tạp

Mỗi giá trị $dp[i]$ chỉ được tính duy nhất một lần, do đó chương trình sẽ có độ phức tạp là $O(n)$.

2. Dãy con tăng dài nhất

Đề bài

Cho dãy số nguyên gồm n phần tử a_1, a_2, \dots, a_n . Một dãy con của dãy số là một cách chọn ra k phần tử bất kỳ của dãy số, với $k \leq n$. Dãy con đơn điệu tăng gồm k phần tử là dãy con $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ thỏa mãn $a_{i_1} < a_{i_2} < \dots < a_{i_k}$.

Yêu cầu: Hãy tìm dãy con đơn điệu tăng dài nhất của dãy đã cho?

Input:

- Dòng đầu tiên chứa số nguyên dương n - độ dài dãy số.
- Dòng thứ hai chứa n số nguyên a_1, a_2, \dots, a_n phân tách nhau bởi dấu cách - các phần tử của dãy ban đầu.

Ràng buộc:

- $1 \leq n \leq 1000$.
- $|a_i| \leq 10^6; \forall i : 1 \leq i \leq n$.

Output:

- Dòng đầu tiên chứa một số nguyên là độ dài của dãy con tăng dài nhất tìm được.
- Dòng thứ hai chứa dãy con đó. Nếu tìm được nhiều dãy thì in ra một dãy bất kỳ.

Sample Input:

```
5
1 4 2 3 2
```

Sample Output:

```
3
1 2 3
```

Phân tích ý tưởng

Hãy gọi $dp[i]$ là độ dài của dãy con tăng dài nhất kết thúc tại phần tử a_i .

Có một nhận xét rằng, trước khi tính $dp[i]$, nếu như ta đã biết được các $dp[0], dp[1], \dots, dp[i-1]$ thì ta hoàn toàn có thể tính được $dp[i]$ bằng cách lựa chọn một vị trí j phía trước nó để ghép a_i vào cuối dãy con đơn điệu tăng kết thúc tại a_j . Điều quan trọng là lựa chọn dãy nào cho tốt nhất?

Nhận xét trên khiến ta rút ra được bài toán này là một bài toán thỏa mãn cấu trúc con tối ưu, và nó lại có bản chất đệ quy. Dĩ nhiên, muốn tạo được dãy con tăng kết thúc tại a_i dài nhất, thì ta cần lựa chọn dãy con tăng dài nhất kết thúc tại một vị trí a_j phía trước a_i , và a_j phải nhỏ hơn a_i (để đảm bảo tính tăng). Từ đây ta rút ra công thức:

$$dp[i] = \max(dp[j]) + 1; \forall j : 1 \leq j < i \text{ and } a_j < a_i$$

Kết quả cuối cùng tất nhiên phải là $\max(dp[i]); \forall i : 1 \leq i \leq n$.

Để truy vết tìm ra dãy con tăng dài nhất, ta sử dụng thêm một mảng $trace[i]$ để lưu lại vị trí của phần tử a_j liền trước a_i trong dãy con tăng dài nhất kết thúc tại a_i . Mỗi khi ta tính được $dp[i] = dp[jmax] + 1$, thì hãy lưu $trace[i] = jmax$. Sau đó, quá trình truy vết diễn ra như sau:

- Gọi $best$ là vị trí có $dp[best]$ lớn nhất. Phần tử cuối cùng được chọn của dãy sẽ là $a[best]$.
 - Phần tử áp chót trong dãy được chọn là $a[trace[best]]$.
 - Phần tử liền trước phần tử áp chót trong dãy được chọn là $a[trace[trace[best]]]$.
- ...

Vậy ta có thể liên tục thực hiện $best = trace[best]$ rồi ghi nhận phần tử ở vị trí $best$, cho tới khi $best = 0$ thì hoàn thành quá trình truy vết.

Cài đặt

Cài đặt C++:

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

using namespace std;

// Nhập dữ liệu.
void enter(int &n, vector < int > &a)
{
    cin >> n;

    a.resize(n + 1);
    for (int i = 1; i <= n; ++i)
        cin >> a[i];
}

// Truy vết tìm ra độ dài dãy con tăng dài nhất.
// và tìm ra một dãy con tăng dài nhất.
void trace_back(int n, vector < int > &a, vector < int > &dp, vector < int > &tr
{
    // Chọn ra vị trí best_pos có dp tại đó lớn nhất.
    int best_pos = 0;
    for (int i = 1; i <= n; ++i)
        if (dp[i] > dp[best_pos])
```

```

    best_pos = i;

cout << dp[best_pos] << endl;

vector < int > lis_elements;
while (best_pos)
{
    lis_elements.push_back(a[best_pos]);
    best_pos = trace[best_pos];
}

for (int i = lis_elements.size() - 1; i >= 0; --i)
    cout << lis_elements[i] << ' ';
}

// Sử dụng công thức truy hồi đê tính bảng phương án.
void solution(int n, vector < int > &a)
{
    vector < int > dp(n + 1), trace(n + 1);

    for (int i = 1; i <= n; ++i)
    {
        // Chọn vị trí jmax tốt nhất để nói a[i] vào sau.
        int jmax = 0;
        for (int j = 1; j < i; ++j)
            if (dp[j] > dp[jmax] && a[j] < a[i])
                jmax = j;

        dp[i] = dp[jmax] + 1; // Cập nhật dp[i].
        trace[i] = jmax; // Lưu vết phân tử đứng trước a[i] là a[jmax].
    }

    trace_back(n, a, dp, trace);
}

main()
{
    int n;
    vector < int > a;

    enter(n, a);
    solution(n, a);

    return 0;
}

```

Cài đặt Python:

```
import sys

# Truy vết và in ra kết quả.
def trace_back(n, a, dp, trace):
    # Chọn ra vị trí best_pos có dp tại đó lớn nhất.
    best_pos = 1
    for i in range(2, n + 1):
        if dp[i] > dp[best_pos]:
            best_pos = i

    print(dp[best_pos])

    # Tìm ra dãy con tăng dài nhất có phần tử kết thúc là a[best_pos].
    lis_elements = []
    while best_pos != 0:
        lis_elements.append(a[best_pos])
        best_pos = trace[best_pos]

    lis_elements.reverse()

    print(lis_elements)

# Tính bảng phương án bằng công thức truy hồi.
def solution(n, a):
    trace, dp = [0] * (n + 1)

    for i in range(1, n):

        # Chọn vị trí jmax tốt nhất để nối a[i] vào sau.
        jmax = 0
        for j in range(1, i - 1):
            if dp[j] > dp[jmax] and a[j] < a[i]:
                jmax = j

        dp[i] = dp[jmax] + 1 # Cập nhật dp[i].
        trace[i] = jmax # Lưu vết phần tử liền trước a[i] là a[jmax].

    trace_back(n, a, dp, trace)

if __name__ == '__main__':
```

```

n = int(input())

a = [0] * (n + 1)
for i in range(1, n + 1):
    a[i] = int(input())

solution(n, a)

```

Đánh giá độ phức tạp

Với mỗi giá trị $dp[i]$, ta cần thực hiện lại một vòng lặp với i thao tác so sánh và kiểm tra để chọn ra vị trí j thỏa mãn $dp[j]$ lớn nhất và $a[j] < a[i]$. Vì thế số lần lặp là $\frac{n \times (n+1)}{2}$, tương ứng với độ phức tạp là $O(n^2)$.

V. Tổng kết

Như vậy, trong bài viết này, chúng ta đã cùng nhau tìm hiểu về những khái niệm cơ bản nhất của phương pháp Quy hoạch động. Hy vọng các bạn đã có thể hình dung được cách hoạt động của phương pháp sau khi đọc những bài toán ví dụ tôi đưa ra.

Ngoài ra, như đã nói ở trên, phương pháp Quy hoạch động thường dùng để giải hai dạng bài toán:

- **Bài toán đếm có bản chất đệ quy.**
- **Bài toán tối ưu có bản chất đệ quy.**

Vì vậy, nếu như các bạn gặp phải một bài toán yêu cầu đếm một đại lượng nào đó, hoặc yêu cầu tìm ra một kết quả **lớn nhất** hay **nhỏ nhất**, thì có nhiều khả năng đó là một bài toán giải bằng phương pháp Quy hoạch động. Điều cần quan tâm cuối cùng là bài toán đó có bản chất đệ quy hay không? Để kiểm tra điều này, các bạn có thể thử suy nghĩ giải bài toán đó bằng Đệ quy (với kích thước nhỏ thì bài toán Quy hoạch động hoàn toàn giải được bằng Đệ quy), rồi mới quyết định có sử dụng Quy hoạch động hay không.

Đôi khi những bài toán có yêu cầu tìm kết quả **lớn nhất** hay **nhỏ nhất** lại là một bài toán sử dụng Tìm kiếm nhị phân thay vì Quy hoạch động, vì thế các bạn cần phân biệt rõ điều kiện cần và đủ của hai phương pháp này, tránh gây ra nhầm lẫn khi giải các bài toán.

Cuối cùng, hãy luyện tập thật nhiều!

VI. Tài liệu tham khảo

- [Competitive Programming 3 - Steven Halim & Felix Halim.](#)
- <https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/>.
- <https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/>.
- [Sách Giải thuật và Lập trình - thầy Lê Minh Hoàng.](#)
- [Tài liệu giáo khoa chuyên Tin quyển 1 - thầy Hồ Sĩ Đàm.](#)