Bài 1. Tổng nhỏ nhất

Giải thuật tầm thường:

Như đã gợi ý trong đầu bài, vét cạn mọi khả năng có thể lựa chọn 4 số **a**, **b**, **c**, **d** trong **k** số x_1, x_2, x_3, …, x_k đã cho,

Với mỗi bộ 4 số đã chọn – tính giá trị thực (kiểu **float**) của các phân số $\frac{a}{b}$ và $\frac{c}{d}$, so sánh và giữ lại kết quả cho tổng $\frac{a}{b} + \frac{c}{d}$ nhỏ nhất.

Với nhóm **Tests I**

Việc vét cạn có thể thực hiện bằng cách khởi tạo tất cả các hoán vị:

- Tạo hoán vị đầu tiên **P** = (1, 2, 3, 4) → p_i=**i**,
- Tạo hoán vị tiếp theo bằng hàm **next_permutation(p,p+4)**,
- Lấy ra 4 số được xác định bởi vị trí của hoán vị, thực hiện các phép xử lý đã nêu.

Phạm vi giá trị các số trong nhóm **Tests I** cho *độ phân giải cao khi chuyển sang số thực*, vì vậy kết quả sẽ chính xác.

Với nhóm **Tests II** có thể thực hiện theo sơ đồ xử lý nêu trên, nhưng cần tính trước và lưu bảng giá trị hoán vị,

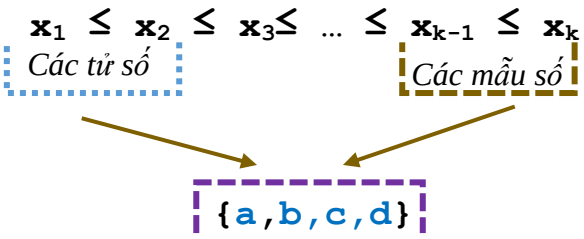
Với nhóm **Tests III** – dùng 4 vòng lặp lồng nhau chọn **a, b, c, d,** kiểm tra, cập nhật kết quả theo yêu cầu nhưng cần dùng số thực với độ chính xác cao hơn – **double**,

Giải thuật tối ưu:

Với tử số và mẫu số đều dương, giá trị phân số tỷ lệ thuận với tử số và tỷ lệ nghịch với mẫu số,

Cần chọn 2 số nhỏ nhất trong các số đã cho làm tử số, hai số lớn trong các số còn lại làm mẫu số.

Như vậy, sắp xếp các số đã cho theo thứ tự tăng dần ta có:



Ta có 3 lựa chọn:

$$\frac{x_1}{x_2} + \frac{x_{k-1}}{x_k} \qquad \frac{x_1}{x_{k-1}} + \frac{x_2}{x_k} \qquad \frac{x_1}{x_k} + \frac{x_2}{x_{k-1}}$$

Dễ dàng chứng minh được rằng cách lựa chọn thứ II cho kết quả nhỏ nhất (việc chứng minh có thể không cần thiết, thay vào đó – tính và so sánh giá trị của 3 biểu thức đã nêu với kết quả kiểu **long double**).

Để lựa chọn vai trò của **a**, **b**, **c**, **d** ta cần kiểm tra $\frac{x_1}{x_{k-1}} - \frac{x_2}{x_k}$.

$$\frac{x_1}{x_{k-1}} - \frac{x_2}{x_6} = \frac{x_1 \times x_k - x_2 \times x_{k-1}}{x_{k-1} \times x_k}$$

Dấu của hiệu cần tìm được xác định bởi biểu thức nguyên ở tử số. Lưu ý là giá trị của biểu thức có thể có bậc 10^{18}.

Độ phức tạp của giải thuật: Với mỗi bộ **k** số - độ phức tạp là $O(k.\log_2 k)$. Với bộ dữ liệu **n** bộ **k** – độ phức tạp sẽ là $O(n.k.\log_2 k)$.

Chương trình ứng với giải thuật tối ưu:

```
#include <bits/stdc++.h>
#define NAME "minsum."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int n,k;
int64_t a[51];

int main()
{
    fi>>n>>k;
    for(int j=0; j<n; ++j)
    {
        for(int i=0;i<k;++i) fi>>a[i];
        sort(a,a+k);
        if(a[0]*a[k-1]- a[1]*a[k-2] <= 0)
            fo<<a[0]<<' '<<a[k-2]<<' '<< a[1]<<' '<<a[k-1]<<'\n';
                else fo<<a[1]<<' '<<a[k-1]<<' '<<a[0]<<' '<<a[k-2]<< '\n';
    }
    //fo<<"\nTime: "<<clock()/(double)1000<<" sec";
}
```

Bài 2. Vô hạn

Với nhóm Tests I:
Trực tiếp khởi tạo dãy 1, 1, 2, 1, 1, 2, 3, 2, . . . và dẫn xuất số ở vị trí cần tìm.
Với nhóm Tests II:
Tìm **k** nhỏ nhất thỏa mãn điều kiện $\mathbf{k^2} \geq \mathbf{n}$,
Nếu viết các đoạn (1), (1, 2, 1), (1, 2, 3, 2, 1), . . . thành từng dòng thì **k** là dòng chứa số cần tìm, từ đó dễ dàng xác định số ở vị trí đã cho.
Với nhóm Tests III: Xử lý tương tự như ở nhóm II, nhưng cần lưu ý làm việc với kiểu **int64_t**.
Với nhóm Tests IV: Cần xử lý số lớn khi tìm **k** nhỏ nhất thỏa mãn điều kiện $\mathbf{k^2} \geq \mathbf{n}$, nhưng do biết k tìm được sẽ thỏa mãn điều kiện $k \leq 10^{16}$ nên không cần xây dựng sơ đồ tổng quát xử lý số lớn.
Giải thuật tối ưu:
Cần kết hợp *triển khai 2 giải thuật* ứng với dữ liệu các nhóm III và IV.
Mỗi số tự nhiên mới trong dãy sẽ tương ứng với một nhóm các phần tử của dãy:

1 → 1

2 → 1 2 1

3 → 1 2 3 2 1

4 → 1 2 3 4 3 2 1

. . . . .

Số **n** sẽ tương ứng với nhóm **2×n-1** phần tử.

Độ dài phần đầu của dãy, khi có **p** nhóm là **1+3+5+. . .+(2×p-1) = p²**.

Để xác định số ở vị trí **n** ta cần biết vị trí cần tìm thuộc nhóm nào.

Gọi **k** là nhóm chứa vị trí **n**.

k sẽ là số tự nhiên nhỏ nhất thỏa mãn điều kiện $\mathbf{k}^2 \geq \mathbf{n}$.

Với $\mathbf{n} \leq 10^{10}$ → bằng cách duyệt trực tiếp có thể tìm **k** với độ phức tạp $O(\sqrt{n})$ (trong phạm vi hạn chế thời gian đủ nhỏ):

```
#include <bits/stdc++.h>
#define NAME "infinity."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int n,k,m,t,ans;

int main()
{
        fi>>n;
        k=1;
        while(k*k<n)++k;
        t=k*k;
        if(t==n)ans=1;
            else
            {
                t=n-(k-1)*(k-1);
                if(t<=k)ans=t; else ans=t+1-k;
            }
        fo<<ans<<'\n';

    fo<<"\nTime: "<<clock()/(double)1000<<" sec";
}
```

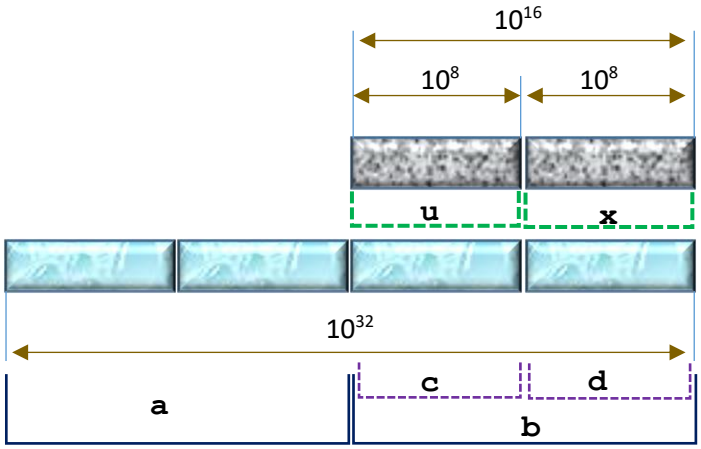
Với $\mathbf{n} > 10^{10}$: *Cần tổ chức tìm kiếm nhị phân.*

Theo điều kiện đầu bài, $\mathbf{n} \leq 10^{32}$ ⇨ $\mathbf{k} \leq 10^{16}$.

Có thể trực tiếp xác định **k** bằng phương pháp tìm kiếm nhị phân. Nhưng việc này đòi hỏi phải tổ chức nhân và so sánh số lớn – một điều khá phức tạp và hiệu quả thực hiện chương trình giảm một cách đáng kể.

Có một cách tiếp cận hiệu quả hơn.

Để tính k^2 ta cần tổ chức biểu diễn k dưới dạng số lớn, trong trường hợp này – hiệu quả nhất là theo cơ số 10^8:



$k = u \times 10^8 + x$

$n = a \times 10^{16} + b$
$= a \times 10^{16} + c \times 10^8 + d$

Dễ dàng tính được u: $u = \lfloor \sqrt{a} \rfloor$.

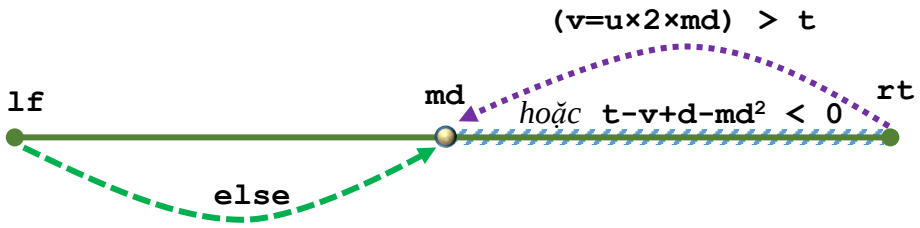
Xác định v: $n = (a, b)$.

$n - u^2 = (a - u^2, b) \leq 2 \times u \times x + x^2$

Lưu ý về phải là một số trong phạm vi biểu diễn của kiểu dữ liệu **int64_t**.

Đặt $t = (a - u^2) \times 10^8 + c$.

Tiêu chuẩn tìm kiếm nhị phân:



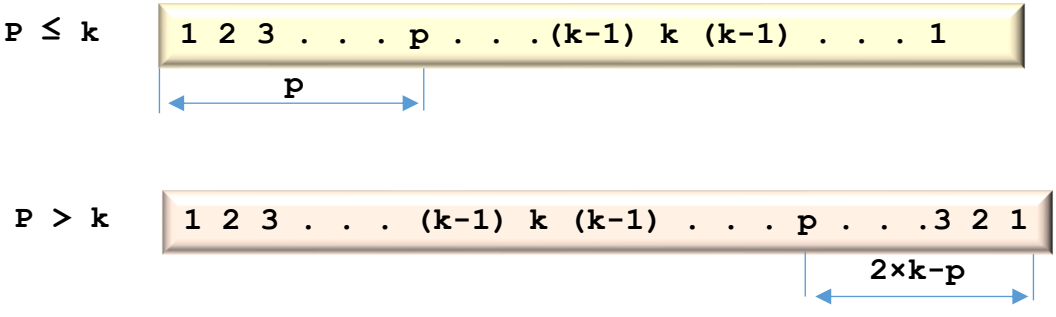
Kết quả nhóm tìm được sẽ là $k = u \times 10^8 + lf$.

Cần tăng kết quả từ tìm kiếm nhị phân lên 1 nếu n không phải là số chính phương.

Vị trí n trong dãy sẽ tương ứng với vị trí p trong nhóm k:

$p = t - v + d - (lf - 1)^2$.

Có 2 trường hợp:

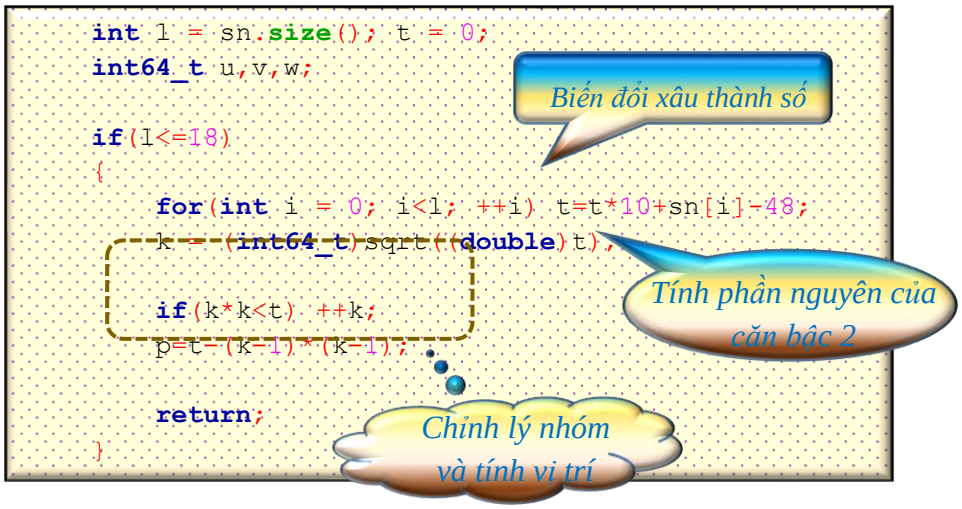


Tổ chức dữ liệu:

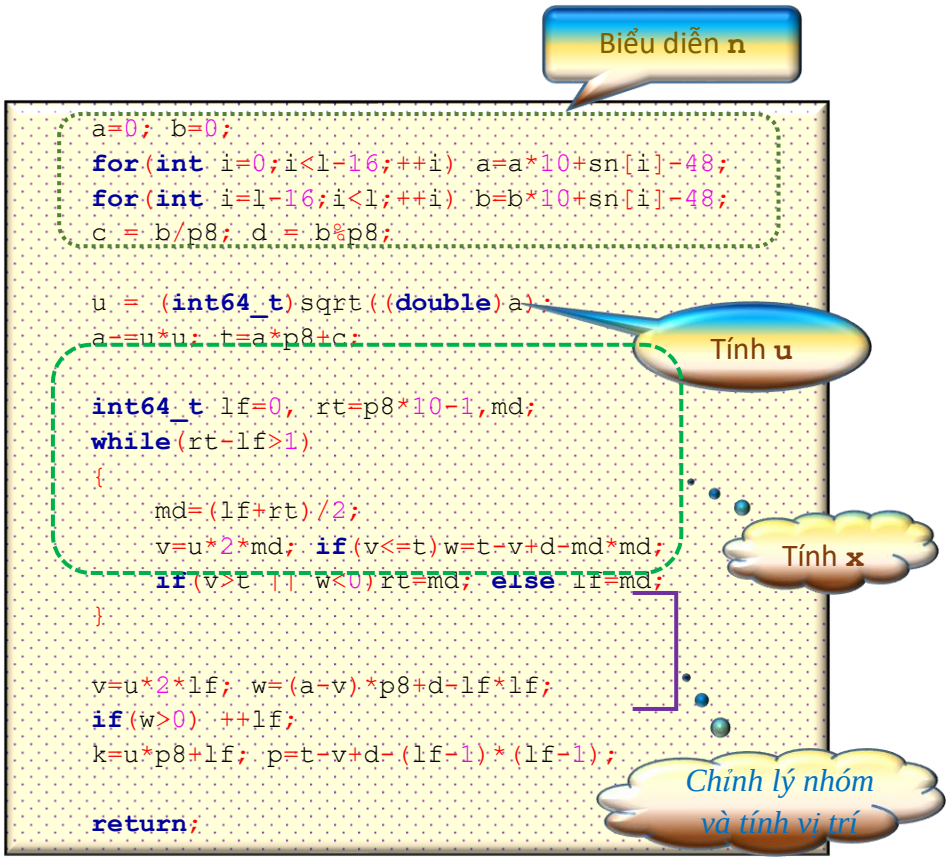
- Lưu trữ **n** dưới dạng xâu,
- Hệ thống các biến đơn **a**, **b**, **c**, **d** – biểu diễn lại **n** phục vụ các phép tính số lớn.

Xử lý:

Với **n** ≤ 10^{18}: trực tiếp tìm kiếm nhị phân xác định nhóm chứa vị trí **n**:



Trường hợp cần làm việc với số lớn:



Nhận xét: Việc triển khai hai sơ đồ xử lý chỉ để giảm thời gian xử lý trong trường hợp cụ thể.

Độ phức tạp xử lý một số n: ≈ O(1).

Độ phức tạp của giải thuật: ≈ O(*m*).

Chương trình

```
#include <bits/stdc++.h>
#define NAME "infinity."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int p8 = (int)1e8;
const int64_t p16 = (int64_t)1e16;
int n,m;
int64_t ans,t,a,b,c,d,k,p;
string sn;

void calc_n()
{
    int l = sn.size(); t = 0;
    int64_t u,v,w;

    if(l<=18)
    {
        for(int i = 0; i<l; ++i) t=t*10+sn[i]-48;
        k = (int64_t)sqrt((double)t);
        if(k*k<t) ++k;
        p=t-(k-1)*(k-1);
        return;
    }

    a=0; b=0;
    for(int i=0;i<l-16;++i) a=a*10+sn[i]-48;
    for(int i=l-16;i<l;++i) b=b*10+sn[i]-48;
    c = b/p8; d = b%p8;
    u = (int64_t)sqrt((double)a);
    a-=u*u; t=a*p8+c;
    int64_t lf=0, rt=p8*10-1,md;
    while(rt-lf>1)
    {
        md=(lf+rt)/2;
        v=u*2*md; if(v<=t)w=t-v+d-md*md;
        if(v>t || w<0)rt=md; else lf=md;
    }
    v=u*2*lf; w=(a-v)*p8+d-lf*lf;
    if(w>0) ++lf;
    k=u*p8+lf; p=t-v+d-(lf-1)*(lf-1);
    return;
}

int main()
{
    fi>>m;
    for(int i=0; i<m; ++i)
    {
```

```
        fi>>sn;
        calc_n();
        if(p>k)p=k*2-p;
        fo<<p<<'\n';
    }
    //fo<<"\nTime: "<<clock()/(double)1000<<" sec";
}
```